InfoPACK User Guide
Version 2.2

NASoftware Limited
1 Prospect Road, Birkenhead
Merseyside CH42 8LE, United Kingdom
Tel: +44(0)151 609 1911

Contents

1 Introduction **2**
- 1.1 Installing and Running *InfoPACK* . . . 3
- 1.2 The Licence . . . 4
- 1.3 How to Contact Us . . . 5
- 1.4 Acknowledgements . . . 6

2 Synthetic Aperture Radar Images **7**
- 2.1 Interpretation of SAR Images . . . 7
- 2.2 The Formation of SAR Images . . . 8
- 2.3 Frequency Bands . . . 9
- 2.4 Polarisation . . . 9
- 2.5 The Number of Looks in an Image . . . 10
- 2.6 The Number of Looks and *InfoPACK* . . . 11
- 2.7 Multi-dimensional Images . . . 11
- 2.8 Correlated Images . . . 12
- 2.9 Reducing the Image Size . . . 12
- 2.10 Sources of SAR Data . . . 12

3 Image File Formats **15**
- 3.1 The *NASoftware* format . . . 15
- 3.2 Built–In File Conversion Facilities . . . 15
  - 3.2.1 File Import: Conversion to *NetCDF* Format . . . 15
  - 3.2.2 File Export . . . 17
- 3.3 General File Formats . . . 17
  - 3.3.1 Using a `.aux` file . . . 17
  - 3.3.2 Using a virtual file . . . 17
  - 3.3.3 File Import Utility . . . 18

4 Image Processing with *InfoPACK* **20**
- 4.1 Segmentation . . . 20
  - 4.1.1 The `segment` routine . . . 21
  - 4.1.2 The `easyseg` routine . . . 24
  - 4.1.3 The `merge` routine . . . 25
  - 4.1.4 The `texseg` routine . . . 26
  - 4.1.5 The `colseg` routine . . . 26
  - 4.1.6 The `defleck` routine . . . 27

4.2 Classification . . . 28
4.2.1 The `classify` routine . . . 28
4.2.2 The `mask_stats` routine . . . 30
4.3 Area Change Detection . . . 31
4.3.1 The `change_detection` routine . . . 31
4.3.2 The `change_detection_position` routine . . . 31
4.4 Detecting Edges . . . 34
4.4.1 The `scanedge` routine . . . 34
4.4.2 The `edge` routine . . . 34
4.5 Speckle Reduction . . . 36
4.5.1 The `despeckle` routine . . . 36
4.6 Point Object Detection . . . 39
4.6.1 The `cfar` routine . . . 39
4.7 References . . . 40
Bibliography . . . 40

5 Using the InfoPACK GUI **41**
5.1 Introduction . . . 41
5.2 Starting The *InfoPACK* GUI . . . 41
5.2.1 Windows . . . 41
5.2.2 Unix/Linux . . . 41
5.3 The Main *InfoPACK* Menus . . . 41
5.3.1 The File Menu . . . 43
5.3.2 The Tools Menu . . . 44
5.3.3 The Segmentation Menu . . . 46
5.3.4 The Classify Menu . . . 47
5.3.5 The Change Detect Menu . . . 49
5.3.6 The Help Menu . . . 49
5.4 The OpenEV Image Viewer . . . 50
5.4.1 Classification Images . . . 50
5.5 Using the *InfoPACK* Shell Window . . . 50
5.5.1 Syntax Highlighting . . . 51
5.5.2 Indentation . . . 51
5.5.3 Command History . . . 51
5.5.4 Function Call Prompting . . . 52
5.5.5 Class Member Prompting . . . 52
5.6 Developing *InfoPACK* Scripts . . . 52
5.7 The IDLE Integrated Development Environment . . . 53

6 Case Studies **55**
6.1 Segmentation and Classification of Intensity Images . . . 55
6.1.1 Introduction . . . 55
6.1.2 Comparison of classification methods . . . 56
6.1.3 Conclusions . . . 61
6.2 Segmentation and Classification of Multiple Intensity Images . . . 62
6.2.1 Introduction . . . 62

6.2.2 Comparison of classification methods . . . 62
6.2.3 Conclusions . . . 66
6.3 Change Detection between Intensity Images . . . 68
6.3.1 Introduction . . . 68
6.3.2 Comparison of classification methods . . . 68
6.3.3 Conclusions . . . 76
6.4 Temporal Signatures of Intensity Images . . . 78
6.4.1 Introduction . . . 78
6.4.2 Classification methods . . . 79
6.4.3 Conclusions . . . 84
6.5 Handling Correlated Images . . . 85
6.5.1 Introduction . . . 85
6.5.2 Correlation and Segmentation . . . 86
6.5.3 Speckle correlation properties . . . 87
6.5.4 Resampling applied to the original intensity image . . . 91
6.5.5 Segmentation: Conclusions . . . 93
6.6 Correlated Images and Despeckling . . . 95
6.6.1 Resampling to remove speckle correlations . . . 96
6.6.2 Conclusions . . . 97
6.7 An Example: Segmenting and Despeckling a Large Image . . . 98
6.7.1 Segmentation . . . 98
6.7.2 Despeckling . . . 101
6.7.3 Conclusions . . . 102
6.8 Segmentation and Classification of Polarimetric Data . . . 105
6.8.1 Introduction . . . 105
6.8.2 Segmentation performance . . . 109
6.8.3 Comparison of classification methods . . . 109
6.8.4 Conclusion . . . 115

A *InfoPACK* High Level Routine Documentation **116**
A.1 User Interface . . . 116
A.1.1 Scripts and Functions . . . 116
A.1.2 GIS Information . . . 117
A.1.3 File I/O . . . 117
A.1.4 Online Help . . . 117
A.2 Common Parameters . . . 118
A.2.1 Image dimensions . . . 118
A.2.2 The Intensity Flag . . . 118
A.2.3 Memory Management Options . . . 119
A.3 Documentation Conventions . . . 119
A.4 High Level Routines . . . 121
A.4.1 `segment` . . . 121
A.4.2 `easyseg` . . . 124
A.4.3 `texseg` . . . 125
A.4.4 `merge` . . . 127
A.4.5 `colseg` . . . 129

A.4.6 defleck . . . 131
A.4.7 classify . . . 133
A.4.8 change_detection . . . 135
A.4.9 change_detection_position . . . 137
A.4.10 despeckle . . . 139
A.4.11 filter . . . 141
A.4.12 filter_bt, filter_det, filter_enl, filter_gorder, filter_korder, filter_mean, filter_var . . . 143
A.4.13 cfar . . . 144

B Low Level and Utility Routine Documentation **146**
B.1 Documentation Conventions . . . 146
B.1.1 pydoc . . . 146
B.2 Image Manipulation Routines . . . 148
B.2.1 add . . . 148
B.2.2 between . . . 148
B.2.3 clip . . . 149
B.2.4 divide . . . 149
B.2.5 edge . . . 150
B.2.6 fill . . . 151
B.2.7 fill_gammarand . . . 151
B.2.8 fill_gaussrand . . . 152
B.2.9 fill_unirand . . . 153
B.2.10 fill_polrand . . . 153
B.2.11 georef . . . 154
B.2.12 layer_sum . . . 155
B.2.13 log10 . . . 156
B.2.14 loge . . . 156
B.2.15 median . . . 157
B.2.16 multiply . . . 157
B.2.17 pca . . . 158
B.2.18 pol2covar . . . 158
B.2.19 polyscan . . . 159
B.2.20 resize . . . 159
B.2.21 scalar_add . . . 160
B.2.22 scalar_divide . . . 161
B.2.23 scalar_multiply . . . 161
B.2.24 scanedge . . . 162
B.2.25 sqrt . . . 163
B.2.26 square . . . 163
B.2.27 subtract . . . 164
B.2.28 sum . . . 164
B.2.29 threshold . . . 165
B.3 File Handling Facilities . . . 166
B.3.1 ascii2nc . . . 166
B.3.2 convert . . . 166

B.3.3 extract . . . 168
B.3.4 gdal_translate . . . 168
B.3.5 gisload . . . 169
B.3.6 gissave . . . 169
B.3.7 insert . . . 170
B.3.8 layer_combine . . . 170
B.3.9 layer_split . . . 171
B.3.10 load . . . 171
B.3.11 makebitmap . . . 172
B.3.12 nanfix . . . 173
B.3.13 nc2ascii . . . 173
B.3.14 nc2raw . . . 174
B.3.15 numload . . . 174
B.3.16 raw2nc . . . 175
B.3.17 save . . . 176
B.3.18 tojpeg . . . 177
B.3.19 totiff . . . 178
B.4 Image Information Routines . . . 180
B.4.1 countseg . . . 180
B.4.2 gdalinfo . . . 180
B.4.3 histo . . . 180
B.4.4 histogram . . . 181
B.4.5 stats . . . 182
B.4.6 xgraphtool . . . 183
B.5 Classification Routines . . . 186
B.5.1 colclass . . . 186
B.5.2 classstats . . . 186
B.5.3 classmap2layers . . . 187
B.5.4 layers2classes . . . 188
B.5.5 mask_stats . . . 188
B.5.6 mask_polstats . . . 190
B.5.7 make_classes . . . 190
B.5.8 matrix_combine . . . 191
B.5.9 matrix_split . . . 192
B.5.10 matrix_print . . . 192

C Library Interface **194**
C.1 Sample Code . . . 194
C.2 Execution . . . 194
C.3 C interface . . . 195
C.3.1 Progress Function . . . 195
C.3.2 Complex Types . . . 196

D *NASoftware* **Conventions for** *NetCDF* **197**
D.1 Dimensions 197
D.2 Variables 197
D.3 Attributes 198
D.3.1 Global Attributes 198
D.3.2 Variable Attributes 199

Chapter 1

Introduction

InfoPACK is a set of tools designed to help in the interpretation of SAR imagery. The toolset incorporates advanced SAR processing algorithms developed by Professor Chris Oliver and colleagues at NASoftware Ltd. These tools use a knowledge of the image formation techniques to remove or take account of the speckle, and are we believe technically well in advance of any other such facilities.

InfoPACK includes routines for segmentation, classification, speckle reduction, edge detection and target detection. These routines accept single or multiple images, including polarimetric and time sequenced image sets. They are fully automatic, allowing reproducible results to be obtained independent of the user, and provide unparalleled facilities for extracting the maximum information from SAR imagery.

There are several easy–to–use methods for using *InfoPACK* routines, ranging from the Graphical User Interface to constructing complex processing chains in an integrated development environment.

This manual provides:

- A summary of the different ways to use *InfoPACK* routines (see this chapter).
- A brief overview of SAR image theory (see chapter 2).
- Information on the native *InfoPACK* image format (see chapter 3).
- Details and commentary on the main *InfoPACK* routines (see chapter 4).
- A detailed guide to the *InfoPACK* GUI (see chapter 5).
- A set of Case Studies illustrating the use of *InfoPACK* (see chapter 6).
- Complete and detailed specification for all the routines (see appendices A and B).

1.1 Installing and Running *InfoPACK*

Installing *InfoPACK*

Hardcopy installation instructions are delivered with the software cdrom. Alternatively installation instructions specific to each supported operating system are in the file `INSTALL.txt` which can be found in the top level installation directory, or alternatively in the top level distribution directory (which is what is delivered in compressed format on the software cdrom).

Running *InfoPACK*

The routines can be accessed in several easy–to–use ways:

From the Graphical User Interface: The GUI provides a point and click environment from which any of the *InfoPACK* routines can be invoked. Successive routines may be invoked in this way. For details on how to invoke the GUI for each supported operating system, see the file `README.txt` which can be found in the top level installation directory.

For each routine, the GUI presents the user with a dialogue box to help specifying the parameters accepted by the routine. For each routine, one or more default parameter sets are available (for example the segmentation routine `segment` provides the choice of `fast, default, sensitive` settings) or users can specify their own choice of parameters.

For the routine specified, the GUI effectively calls the corresponding Python script; the basic script commands are echoed on the console so that you can see what is going on. In this way, user scripts specifying complex processing chains can be built up experimentally and "on the fly". Operations are file to file, and memory is cleared after the operation is complete.

Operation of the GUI is the subject of chapter 5.

From a Python script: Single routines may instead be called from the system command line prompt. Users can thus develop complete Image Understanding processing chains, of both *InfoPACK* and other routines, by composing appropriate shell or batch scripts.

For details on how to invoke command line scripts under each supported operating system, see the file `README.txt` which can be found in the top level installation directory.

From a Python shell or program: All *InfoPACK* routines can be invoked as Python function calls from within a Python shell or Python program. The routine procedures are automatically made available from the module `infopack` and may be called directly or from within the users own script. When called directly from the console, online reminders are provided for the required parameters.

Python provides an integrated development environment for researching and developing complete Image Understanding processing chains including calls to *InfoPACK* routines.

Sample data is included with the InfoPACK distribution in the `images` subdirectory of the main installation directory. Also located there is a `README` file that explains in some detail how to process the sample data with several different routines.

More details on running *InfoPACK* and commentary on the main routines can be found in chapter 4. Appendices A and B contain exact syntaxes for the Python Scripts and Functions including descriptions of all parameters and their default values.

1.2 The Licence

The Software

'The Software' means the *InfoPACK* toolset developed by *NASoftware Ltd* including all supporting software and documentation supplied with it.

Your Rights as Purchaser

The purchase of this manual and accompanying software from *NASoftware Ltd* or from an authorised dealer confers a licence to use the software on one machine (single-user licence) or simultaneously by a number of users (multi-user licence). You may make as many backup copies as you need for security, and use the software on any suitable machine. Apart from this right to make backup copies, all copyright and rights of reproduction are reserved. No part of this or other documentation may be reproduced or used in any form or by any means, including photocopying, without the prior written consent of *NASoftware Ltd.*

Please keep the original media as proof of purchase and quote the product name and version number in any correspondence.

Your Licence

The licence to use the software does not include a licence to resell all or any part of it, either in the form in which you received it or as a subsystem of any other software package. Please contact *NASoftware Ltd* for details on run-time licences.

Warranty

NASoftware Ltd gives no warranty that all errors have been eliminated from this manual or from the software supplied with it. Nor does *NASoftware Ltd* give any warranty or representation as to the fitness of the software for any particular purpose. We strongly recommend that before using the software for any valued purpose you test its suitability for that purpose. *NASoftware Ltd* disclaims all responsibility for any direct, indirect or consequential losses, damages, costs, expenses or claims of any nature resulting from any deficiency in either the documentation or the software.

1.3 How to Contact Us

Getting Help

If you have problems installing our software then please contact us. We can also provide suggestions on how to use *InfoPACK* to get the best results from your images.

To request help via e-mail send a message to

```
support@nasoftware.co.uk
```

It may also be worth checking the information on our website

```
http://www.nasoftware.co.uk/
```

Bug Reports

If you have any problems or find bugs in this software, please inform us. Give the following information:

- Version of the software (run any command-line routine with the `-v` option).
- Platform and operating system.
- The commands which produced the error, and a description of the behaviour.
- If possible, the images involved so that we can try to reproduce your results.

To report bugs via e-mail send a message to

```
support@nasoftware.co.uk
```

Other Addresses

You can also contact us by telephone, fax or mail.

NASoftware Ltd
1 Prospect Road
Prenton
Merseyside
CH42 8LE
United Kingdom

Phone: (+44) 151 637 1422
Fax: (+44) 151 550 7830

1.4 Acknowledgements

We are grateful to:

- the National Environmental Research Council for permission to make use of some of their very nice DLR ESAR high resolution SAR data for some of the Case Studies in this documentation.
- Dr Rod Cook and N.A. Software Ltd for extensive discussions during the design and implementation of *InfoPACK*.
- The many users whose comments and encouragement have helped develop Release 2.

Chapter 2

Synthetic Aperture Radar Images

A radar system illuminates an area with microwaves, and records the strength and travel-time of the returned signals. This allows the distance (or range) of the reflecting objects to be determined. As in an optical instrument, the resolution of such a system is affected by the size of the aperture: a larger aperture gives a finer resolution. However, scaling up to radar wavelengths has problems. For example, to achieve an (azimuth) resolution of 100m with a 5cm radar from a range of 800km requires an aperture some 400m long.

If the radar is attached to a moving platform, either a satellite or an aircraft, then it is possible to combine reflected signals from along the flight path to synthesise a very long antenna. The aperture, or area used to receive signals, is created artificially during the signal processing. This is a Synthetic Aperture Radar (SAR).

2.1 Interpretation of SAR Images

Each pixel in a SAR image represents the back-scattered radiation from an area in the imaged scene. A large pixel value (bright) represents a strong received signal. The strength of the received signal depends on many things. These include:

- Radar wavelength and the relative size of the scatterers.
  Smooth, flat surfaces usually reflect very little radiation back to the antenna, so roads and still water will appear dark. Objects much smaller than the wavelength of the radar are invisible to the detector, so for a low frequency (L-band) radar, fields may also appear smooth. At shorter wavelengths, fields or woodland often give the image a textured appearance.

- Polarisation of the pulses.
  Radar pulses can be polarised into a vertical or horizontal plane. For a linear feature on the ground, the reflected signal is much stronger when the polarisation agrees with the orientation of the scatterer. For example, a pylon will appear much brighter in a VV[1] image than in an HH image. The same is true for other objects with a strong vertical component such as walls, and waves.

  For larger surface features, the changes are unpredictable. However, the relative intensities in a patchwork of fields can be quite different under different polarisations, allowing field

[1] See Section 2.4 for a description of this notation.

boundaries invisible in one image to be detected in another; or providing the means to distinguish between various classes of ground cover much more reliably than if only unpolarised data is available.

- Topography of the ground.
  Ground which is sloping towards the antenna will appear brighter than ground sloping away. This effect is known as **layover**. It is also possible to get radar shadows — dark areas in an image that were not illuminated.

- Water content of the imaged area.
  The scattering of radar pulses depends on the dielectric properties of the scatterers. These are affected by water: drier objects appear darker than wetter ones (except that, as noted previously, still surface water appears dark).

- Incidence angle.
  The incidence angle is the angle between the radar beam and a perpendicular on the imaged area. A small incidence angle gives high backscatter; larger angles give weaker signals but larger shadows, so increasing the texture in the image. Images taken at large incidence angles are also less susceptible to layover.

- Direction of the flight-path.
  The appearance of linear features such as walls, hedges and ocean waves will depend on their orientation relative to the radar beam. If buildings are aligned to the beam, it is possible to get a 'double bounce' of the radar — once off the building, and once off the ground. This produces very bright pixels in the image. To a lesser degree, this also happens from tree-trunks along the edge of a wood.

  These effects are also present in satellite SAR imagery.

2.2 The Formation of SAR Images

A SAR system illuminates a scene with microwaves and records both the amplitude and the phase of the back-scattered radiation, making it a coherent imaging process. An inherent feature of SAR imaging is **speckle** which arises because each resolution cell contains many scatterers; the phases of the return signals from these scatterers are randomly distributed and speckle is caused by the resulting interference.

The received signal is sampled and converted into a digital image. Note that the pixel spacing in the final image and the resolution of the system may not be the same. If a system with a resolution of 5m is set to oversample the signal and produces an image in which each pixel corresponds to 3m, you do not get a more detailed image — the information is just smeared out over several pixels. In this case, the image is **correlated** and is much more difficult to process.

The field recorded at pixel x, denoted $E(x)$, can be written

$$E(x) \;=\; \sum_{s} a(s)\;\exp(i\,\phi(s))\;h(s,x)$$

where the summation ranges over the scatterers, $a(s)$ and $\phi(s)$ are the amplitude and phase (respectively) of the signal received from scatterer s, and h is the instrument (or point-spread) function. The value of h is near 1 when s is in or near the resolving cell corresponding to pixel x, and near

zero otherwise. If we assume that h is translation-invariant (does not depend on x) then it can be written as a one-parameter function: $h(s-x)$.

Thus the detected field E is an array of complex numbers. The square of the modulus of the field at x is called the detected **intensity** at x; the square-root of the intensity is called the **envelope** or the **amplitude**. Note that this is not the same as the amplitude of the received signal because the received field is perturbed by the instrument function. The amplitude of the received signal, $a(s)$, is called the **reflectivity**, and its square is called the **surface cross-section**. It is this that we try to recreate when reducing the speckle.

Datasets for SAR images can be stored as either intensity or amplitude data. It is important that you know what type of data you are using. Many images are stored as amplitude since the speckle is less obtrusive when these images are displayed.

2.3 Frequency Bands

The microwave part of the electromagnetic spectrum (1–30cm) is used by radar. It was broken up into bands, which were given arbitrary labels during World War II. These names have entered radar jargon and are commonly used. They are listed in the table in Figure 2.1. For wavelengths in the region 30–75cm, the term P-band is often used. Because of atmospheric absorption effects, only certain sharply defined windows in the spectrum can be used for imaging purposes.

Band	Frequency Range	Wavelength Range
P	0.4–1 GHz	30–75cm
L	1–2 GHz	15–30cm
S	2–4 GHz	7.5–15cm
C	4–8 GHz	3.7–7.5cm
X	8–12 GHz	2.5–3.7cm

Figure 2.1. The IEE names for radar frequency bands.

2.4 Polarisation

When the radar transmits microwaves, they are polarised into a vertical or horizontal plane. The antenna also detects the scattered signal in a particular polarisation — not necessarily the same as that transmitted. This gives four possible images of the same scene. They are labelled HH, HV, VH and VV for Horizontal/Vertical <transmit>–<receive>. However, if the SAR is correctly calibrated, the backscatter reciprocity principal implies that the HV and VH images are the same.

Instruments such as the JPL AirSAR can image a scene in two orthogonal polarisations simultaneously. This is known as **quadrature polarisation**. It enables the complete scattering matrix to be computed. Other instruments, such as the Envisat ASAR system, can produce data for both VV and HH polarisations, though not the complete scattering matrix. Fully polarimetric data will also be available from future planned satellite SAR instruments.

Whenever more than one polarisation is available, it is likely that improved results can be achieved by processing both/all polarisations jointly.

2.5 The Number of Looks in an Image

The number of looks in an intensity image is a measure of the statistical fluctuations introduced by speckle resulting from the interference between randomly-positioned scatterers. It is purely a property of the imaging process and depends on the number of independent speckle estimates that have been combined to form the final image. For example, ERS PRI data is described as 3 look. This means that the final image has been formed by adding the intensities from three independent single-look complex (SLC) components.

To reduce speckle levels even further, many people sacrifice the resolution of a single image and produce a smaller one in which each pixel is formed by averaging a clump of four pixels in the original (say). If the image is uncorrelated, this process increases the number of looks by a factor of four. Unfortunately, there is usually little redundant information in a SAR image (unlike the situation in optical images) because objects such as trees, vehicles and buildings typically occupy only at most one or two pixels (for currently available or envisaged satellite SAR systems).

The interaction between the ENL, the pixel spacing and the resolution must be understood. Let us first define these quantities.

Effective (Equivalent) Number of Looks The statistics of the speckle are usually described in terms of the Effective (or Equivalent) Number of Looks (ENL). This is intended to represent the speckle component of the observed intensity. Normally, of course, the intensity combines both speckle and RCS fluctuations. However, if the underlying RCS is uniform the ENL can be defined as

$$\text{ENL} = \frac{\text{mean}^2}{\text{variance}}$$

In general estimating the looks measure yields values which are smaller than the true ENL because additional RCS fluctuations are included. Only if the RCS is constant, or if the observed intensity is divided by the true underlying RCS, would the looks measure correspond to the ENL.

Resolution The physical characteristics of the radar determine how far apart two objects on the ground must be in order for them to be distinguished in the image. This minimum distance is the spatial resolution of the image. The resolution in the range (across-track) direction is determined by the bandwidth of the transmitted pulse: larger bandwidth equals higher resolution. The resolution in the azimuth (along-track) direction is determined by the length of the radar antenna: a shorter antenna gives a higher resolution.

Pixel spacing The pixel spacing in an image is determined by the sampling frequency of the processor. An image can be oversampled so that the pixel size is smaller than the resolution. When this happens, the values of neighbouring pixels are correlated. To obtain independent samples the image must be averaged down until the pixel spacing is at least as large as the resolution.

If we now consider averaging the intensity in neighbouring correlated pixels the ENL will not be reduced by the expected amount because the speckle is correlated. ERS PRI data is described as having an ENL of 3.0. The pixel size is about $12.5\text{m} \times 12.5\text{m}$ and the resolution about $25\text{m} \times 25\text{m}$. Thus averaging 2x2 pixels does not result in an ENL of 12 but a value nearer 6.

2.6 The Number of Looks and *InfoPACK*

Many of the *InfoPACK* routines, for example the utility stats.py (or from the GUI, select Tools/Image Statistics), estimate the number of looks from data using the definition of ENL above. Since this estimate normally contains both RCS and speckle fluctuations this estimate only corresponds to the ENL when there are no RCS variations. We describe the result as the ”looks measure” to distinguish it from the true ENL.

Occasionally, InfoPACK routines accept the number of looks as an input. In this case the defined instrument value, eg. 3 for full resolution ERS PRI, should be used.

Even if you do know the number of looks in your data, you may still get better results by allowing the InfoPACK functions to estimate it from the image. The *InfoPACK* functions use the looks parameter to build a model of the surface cross-section that underlies the speckled image. When the observed intensity is divided by the reconstruction the result resembles pure speckle so that the looks measure is equivalent to the ENL. It is important to get a good model, and the functions are tuned to work with their own estimates. The estimates are usually slightly different from the ‘real’ value. They are close when the image consists mainly of fields, and get poorer when estimated from a town scene.

2.7 Multi-dimensional Images

A multi-dimensional image is a composite of several images of the same scene. The component images are registered so that equivalent pixels correspond to the same location on the ground.

A three-dimensional image is often displayed as a colour-composite in which the three individual images correspond to the red, green and blue components. Variations in colour reflect differences between the images caused by changes on the ground such as growth of vegetation, or change in water content.

Multi-dimensional images can be generated in various ways.

Complex Data. The *NetCDF* image format treats complex data by storing the real and imaginary parts in separate layers; hence, formally a complex dataset is stored in a multi–dimensional image. We take this for granted below; but (for example) complex data is usually stored for the separate polarisations in a multi-polarimetric image.

Multi-temporal images. The images are taken at different times, possibly months apart. This allows temporal changes to be monitored.

Multi-frequency images. The images are formed with different frequency radars. Since details smaller than the wavelength of the beam are invisible, a multi-frequency image shows structures on different length scales.

Multi-polar images. Like light, radar can be polarised into horizontal (H) and vertical (V) planes. Transmitting H or V polarised radiation and receiving H or V gives four possible images of the scene. Multi-polar images are also called **polarimetric** images.

Multi-sensor images. Remote sensing images cover a large range of the electromagnetic spectrum: visible, infra-red, thermal radiation, and microwaves. Images from several bands can be selectively combined to discriminate between particular features of interest. *InfoPACK* makes it straightforward to carry out data-fusion processing in this way.

2.8 Correlated Images

All of the routines in *InfoPACK* assume that the pixel values in an image are statistically uncorrelated. If this requirement is not satisfied, (you are using ERS-1 PRI data, for example) the performance of the routines can be affected.

Two major sources of correlations are:

- Oversampling the signal during the image formation process (making the pixel size less than the spatial resolution).
- Transforming the image so that the pixel lattice is aligned with latitude and longitude lines of the Earth. This process, known as **geo-coding**, can also alter the statistical properties of the image. We recommend geo-coding *after* processing the image.

To overcome the problems caused by oversampling, the image should be resampled before being processed. Section 6.5 gives a detailed discussion of the effect of correlation, and how to remove it from both intensity/amplitude and complex single look imagery, both when the degree of oversampling is known and when it is not known a priori but must be estimated from the data.

2.9 Reducing the Image Size

Many of the algorithms in *InfoPACK* make use of a physically correct model derived from the image-formation process that produces SAR images. This is particularly true of the speckle-reduction and edge-detection functions. The more the statistical properties of an image deviate from those of a SAR image, the less optimally these algorithms will perform. *Because of this, it is important to resize or average down images as intensity data and not as amplitude data.* We provide the utilities (see appendix B) to help you do this.

2.10 Sources of SAR Data

Shuttle Imaging Radar (SIR) experiments

SIR-A flew on the space shuttle ‘Columbia’ and collected data on 12 November, 1981. It is L-band, HH polarised.

National Space Science Data Center
Coordinated Request User Support Office
NASA/Goddard Space Flight Center
Code 633, Building 26, Room 113
Greenbelt. MD 20771
USA

Tel: (+1) 301 286 6695
Fax: (+1) 301 286 1771
URL: `http://nssdc.gsfc.nasa.gov/earth/earthfaq.html`

SIR-B flew on the space shuttle 'Challenger' and collected data on 5 October, 1984. It is L-band, HH polarised.

SIR-C has flown on several space shuttle missions since 1993. It has L-band and C-band SAR with quadrature polarisation, and also X-band, HH polarised.

Radar Data Center
Jet Propulsion Laboratories
California Institute of Technology
4800 Oak Grove Drive
Pasadena. CA 91109
USA

Fax: (+1) 818 393 2640
URL: http://southport.jpl.nasa.gov/

AirSAR

An airborne SAR which collects multi-frequency data at C, L, and P bands.

Radar Data Center
Jet Propulsion Laboratories
California Institute of Technology
4800 Oak Grove Drive
Pasadena. CA 91109
USA

Fax: (+1) 818 393 2640
URL: http://southport.jpl.nasa.gov/

Japanese Earth Resources Satellite

JERS-1 (Fuyo 1) was launched on 11 February, 1992. It is L-band, HH polarisation, with $38.5°$ incidence angle.

National Space Development Agency of Japan (NASDA)
Earth Observation Program Office
Tokyo 105
JAPAN

Tel: (+81) 3 5470 4254
URL: http://hdsn.eoc.nasda.go.jp/

European ERS Satellites

ERS-1 was launched by the European Space Agency (ESA) in July 1991. It is C-band, VV polarisation, with 23° incidence angle.

ERS-2 was launched in 1995 to continue the work.

ERS Helpdesk
ESRIN
Via G. Galilei
00044 Frascati
Rome
ITALY

Tel: (+39) 6 941 80666
Fax: (+39) 6 941 80272
URL: http://www.esa.int/

European Envisat ASAR

Envisat was launched in 2002; the C-band ASAR instrument provides 25m resolution and H and V polarisations, with better coverage, range of incidence angles, and modes of operation than ERS.

ERS Helpdesk
ESRIN
Via G. Galilei
00044 Frascati
Rome
ITALY

Tel: (+39) 6 941 80666
Fax: (+39) 6 941 80272
URL: http://www.esa.int/envisat/instruments.html

Radarsat

Radarsat is a Canadian satellite launched in August, 1995. It is C-band, HH polarisation, with a 20°–60° variable incidence angle.

Radarsat International, Inc.
275 Slater Street
Suite 1203
Ottawa
Ontario, K1P 5H9
CANADA

Tel: (+1) 613 238 5424
Fax: (+1) 613 238 5425
URL: http://radarsat.space.gc.ca/

Chapter 3

Image File Formats

There are a wide variety of file formats currently used for storing image data and it would be impracticable for *InfoPACK* to support them all. Instead *InfoPACK* has chosen to support the Geospatial Data Abstraction Library (GDAL found at `http://www.remotesensing.org/gdal/`) which is a translator library for raster geospatial (GIS) data formats. GDAL currently supports roughly 40 format drivers, some of which are listed in Table 3.1.

If your image is in one of the formats supported by GDAL then you should be able to run any of the *InfoPACK* routines on your data directly. Alternatively, your existing GIS software may be able to export the data into one of the formats supported by GDAL.

If you are still unable to process your data with *InfoPACK* then there are a variety of methods that may be used to convert the data.

3.1 The *NASoftware* format

InfoPACK uses a simplified version of the Unidata *NetCDF* (`http://www.unidata.ucar.edu/packages/netcdf/`) file format for its default format. *NetCDF* is a general purpose format that allows *InfoPACK* to save intermediate results without loss of accuracy. However full use of this flexibility would actually complicate the storing of single and layered images, so we use a subset of the functionality, known as the *NASoftware* format. If your software allows you to write *NetCDF* files, then see Appendix D for details of how to write *NetCDF* files in the *NASoftware* format.

3.2 Built–In File Conversion Facilities

All of the *InfoPACK* routines will read any of the formats supported by GDAL, but not all of these support writing so the default is to write images in the *NASoftware* format.

NetCDF files can be converted to other standard image formats using the `Export Image` item on the main `File` menu of the *InfoPACK* GUI; see section 5.3.1 for more details.

3.2.1 File Import: Conversion to *NetCDF* Format

Conversion to *NetCDF* format is provided via the `Import` item on the main `File` menu. This supports a wide variety of standard image formats. You will be prompted for the file to be imported

Long Format Name	Code	Creation	Georeferencing
Arc/Info ASCII Grid	AAIGrid	Yes	Yes
Arc/Info Binary Grid (.adf)	AIG	No	Yes
Microsoft Windows Device Independent Bitmap (.bmp)	BMP	Yes	Yes
BSB Nautical Chart Format (.kap)	BSB	No	Yes
CEOS (Spot for instance)	CEOS	No	No
First Generation USGS DOQ (.doq)	DOQ1	No	Yes
New Labelled USGS DOQ (.doq)	DOQ2	No	Yes
Military Elevation Data (.dt0, .dt1)	DTED	No	Yes
Eosat Fast Format	EFF	No	No
ERMapper Compressed Wavelets (.ecw)	ECW	Yes	Yes
ESRI .hdr Labelled	EHdr	No	Yes
ENVI .hdr Labelled Raster	ENVI	No	Yes
Envisat Image Product (.n1)	Envisat	No	No
EOSAT FAST Format	FAST	No	Yes
FITS (.fits)	FITS	Yes	No
Graphics Interchange Format (.gif)	GIF	Yes	No
Arc/Info Binary Grid (.adf)	GIO	Yes	Yes
GRASS Rasters	GRASS	No	Yes
TIFF / GeoTIFF (.tif)	GTiff	Yes	Yes
Hierarchical Data Format Release 4 (HDF4)	HDF4	Yes	Yes partially
Erdas Imagine (.img)	HFA	Yes	Yes
NASoftware NetCDF Images (.nc)	*NASoftware*	Yes	Yes
Atlantis MFF2e	HKV	Yes	Yes
Japanese DEM (.mem)	JDEM	No	Yes
JPEG JFIF (.jpg)	JPEG	Yes	Yes
JPEG2000 (.jp2, .j2k)	JPEG	Yes	Yes
NOAA Polar Orbiter Level 1b Data Set (AVHRR)	L1B	No	Yes
Atlantis MFF	MFF	Yes	Yes
NITF	NITF	Yes	Yes
OGDI Bridge	OGDI	No	Yes
PCI .aux Labelled	PAux	Yes	No
Portable Network Graphics (.png)	PNG	Yes	No
Netpbm (.ppm,.pgm)	PNM	Yes	No
USGS SDTS DEM (*CATD.DDF)	SDTS	No	Yes
SAR CEOS	SAR_CEOS	No	Yes
USGS ASCII DEM (.dem)	USGSDEM	No	Yes
X11 Pixmap (.xpm)	XPM	Yes	No

Table 3.1. File Formats supported by GDAL and *InfoPACK*.

and for the converted filename (default suffix `.nc`). The input file format is taken from the input filename suffix. If the format is recognised, the data will be converted to *NetCDF* format in the output file.

You will not normally need to import non–*NetCDF* files before processing them; they will be loaded provided that their format is recognised.

3.2.2 File Export

NetCDF files can be converted to other standard image formats using the `Export Image` item on the `Tools` menu; see section 5.3.1 for details.

Alternatively, the routine `gdal_translate` may be run from a command line. Specific routines to output raw data (`nc2raw.py`), ascii (`nc2ascii.py`) and TIFF bitmap (`totiff.py` and `makebitmap`) files are also provided. Details of these routines and, in particular, the supported file formats are given in Appendix B.

3.3 General File Formats

If your file format is not supported by *InfoPACK*, but you can describe how the data is laid out within the file, then GDAL supports two different methods for reading the data.

3.3.1 Using a `.aux` file

The first method is a partial implementation of the PCI `.aux` labelled format. For this, a text file with the file extension `aux` is created which describes the image layout, for example:

```
AuxilaryTarget: out.raw
RawDefinition: 512 512 1
ChanDefinition-1: 8U 0 1 512 Swapped
UpLeftX: 440720.000
UpLeftY: 3751320.000
LoRightX: 471440.000
LoRightY: 3720600.000
```

All PCI data types (8U, 16U, 16S, and 32R) are supported. Currently georeferencing, projections, and other metadata is ignored.

The implementation of GDAL allows for read, write and creation. To open a PCI labelled file, select the raw data file itself. The .aux file (which must have a common base name) will be checked for automatically.

For more details of the PCI aux format see
`http://www.remotesensing.org/gdal/frmt_various.html`

3.3.2 Using a virtual file

A second approach to describing your file format would be to create a virtual file. For example a `vrt` file with two bands that happen to both come from band one of another file would be:

```
<VRTDataset rasterXSize="512" rasterYSize="512">
   <VRTRasterBand dataType="Byte" band="1">
     <SimpleSource>
       <SourceFilename>utm.tif</SourceFilename>
       <SourceBand>1</SourceBand>
       <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
       <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
     </SimpleSource>
   </VRTRasterBand>
   <VRTRasterBand dataType="Byte" band="2">
     <SimpleSource>
       <SourceFilename>utm.tif</SourceFilename>
       <SourceBand>1</SourceBand>
       <SrcRect xOff="0" yOff="0" xSize="512" ySize="512"/>
       <DstRect xOff="0" yOff="0" xSize="512" ySize="512"/>
     </SimpleSource>
   </VRTRasterBand>
</VRTDataset>
```

3.3.3 File Import Utility

If the built-in import facilities do not cover your file format you can use the general conversion routine `raw2nc.py` provided to convert your images into *NetCDF* format. This routine is described in detail in Section B.3.16. There is also an export routine `nc2raw.py`. To use `raw2nc.py` you will need to specify:

-l n The number of image layers (default 1)

-tC the type of data in the raster file There are eight recognised types:

option -t l	default integers (signed)
option -t 1	Int0, Int8 (signed)
option -t s	2-byte integers Int16 (signed)
option -t i	4-byte integers Int32 (signed)
option -t d	Float, Float64
option -t f	Float0, Float8, Float16, Float32
option -t F	Complex0, Complex8, Complex16, Complex32
option -t D	Complex, Complex64

-b Byteswap.

Depending on how your images are constructed, it is possible that you will need to reverse the order of the bytes. To see whether this is the case, we suggest you load the converted image into the viewer and use the `Image Statistics` menu button to check that the values it reports look sensible; if not, you probably need to swap the byte order.

If you do need to swap the byte order, include the `-b` flag.

option -s reverse the byte order.

-width=nw -height=nh The image size.

The dimensions in the image are measured in pixels.

option `-width` the number of pixels per row of the image

option `-height` number of rows in the subimage (you can import only some of the rows)

Chapter 4

Image Processing with *InfoPACK*

This chapter illustrates how to use the main *InfoPACK* routines, and some of the results that can be achieved, to show what they do and how the parameters affect the results. Lengthier Case Studies, geared more towards specific applications, are given in Chapter 6.

Many of the test images used in this chapter are simulated 3–look SAR data, but *InfoPACK* can be used on SAR data from any source and with any size of looks measure, and also on optical images: see section 6 for examples. Sample data is included with the InfoPACK distribution in the `images` subdirectory of the main installation directory. Also located there is a `README` file that provides in some detail how to process the sample data with several of the routines described in this chapter.

4.1 Segmentation

Speckle is the most characteristic feature of SAR imagery, and the removal of speckle is an important step in the automatic interpretation of SAR images. Many techniques have been proposed for reducing the level of speckle. These include simple averaging and applying standard filters. However, these methods tend to blur edges and other small features, reducing the sharpness of the detail, and hence the usefulness of the processed image.

The algorithms provided in *InfoPACK* use the known statistical properties of SAR data derived from a theoretical analysis of the image formation process. By using a physically correct model they are able to preserve the detail and resolution of the original image while producing a smooth, real-valued output.

All the speckle-reduction algorithms in *InfoPACK* work in a similar way: they solve an optimisation problem to find the best fit of a given model to the image data. They differ in the kind of model which is fitted, or the kind of data (single or multi-dimensional images) accepted.

The most common need is not *only* to remove speckle; it is to remove it in order to split the image into homogeneous regions (segmentation) or classes (classification).

Segmentation yields a number of regions consistent with the statistical uncertainty, the constraint and the data model. Classification is a much simplified representation in terms of useful categories, such as field with crop. Segmentation can thus provide a wealth of information about how the measured parameter varies within a single class.

Here we describe how to use the main segmentation facilities provided.

4.1.1 The `segment` routine

The routine `segment` segments a single intensity, amplitude or complex image; or a complex polarised image; or a sequence (eg a multi-temporal set) of images. Ideally a segmentation algorithm would divide up an image into regions, each of which represented a feature of interest (a wood, town or field, for example). In practice, they respond to qualities in the image, such as variations in intensity or texture, which may or may not correspond with your areas of interest.

The `segment` segmentation algorithm produces **cartoon** images: it looks for regions of constant underlying RCS (allowing for speckle) and produces an output image in which each region in the resulting segmentation is coloured with the mean of the data values in that region (or determinant values in the case of polarimetric images).

Although the data values in a cartoon form a discrete set, a segmentation is not the same as a classification: the number of regions may be in the thousands, but these regions could normally be classified into a small number of categories.

The `segment` algorithm accepts one or more images; multiple images are used together to produce a jointly optimal set of segments. The algorithm starts with a rectangular segmentation of the image(s). The initial size of the rectangles is determined by the user (setting the average region size in pixels); overall, the default (start with single pixels) is suitable.

Pixels located at the edges of regions are transferred from one to another. The likelihood that the image data fits a given segmentation can be calculated, so it is possible to judge whether the transfer of a pixel improves the segmentation. The calculation also takes into account the straightness of the region boundaries: the amount of crinkle allowed is set by the user.

This process of transferring pixels is governed by simulated annealing, and it will eventually find the global optimal solution for the segmentation of the image into a specified number of regions. An optional post-processing stage is available (routine `merge`) which merges regions which are statistically similar.

Full Python script and function syntaxes are given in section A.4.1.

An Example Segmentation

Figure 4.1 shows a simulated 3–look SAR image (this image is included with the InfoPACK distribution in the `images` directory). An important issue with applying segmentation is the selection of operating parameters. There are three main ones that control the process.

1. The shape parameter attaches a penalty to excessive curvature in the reconstruction. Large values of the parameter allow many changes of direction of the edge. Small values bias the reconstruction towards smoother edges. Since speckle yields very crinkled edges to regions of similar intensity it is important to adopt a large enough curvature penalty, i.e. a small enough shape parameter.

2. The optimisation method has two further parameters attached. The first is the stopping parameter to assess when the reconstruction is complete. For noisier data one would expect to need this parameter to be increased. Smoother data, such as that encountered on averaging speckle (eg via multi–looking), should allow a reduced value of the stopping parameter.

3. The second parameter in the annealing method is the cooling parameter which controls the rate at which the optimisation converges on the final solution. If this is made too large the solution may freeze out away from the optimal result.

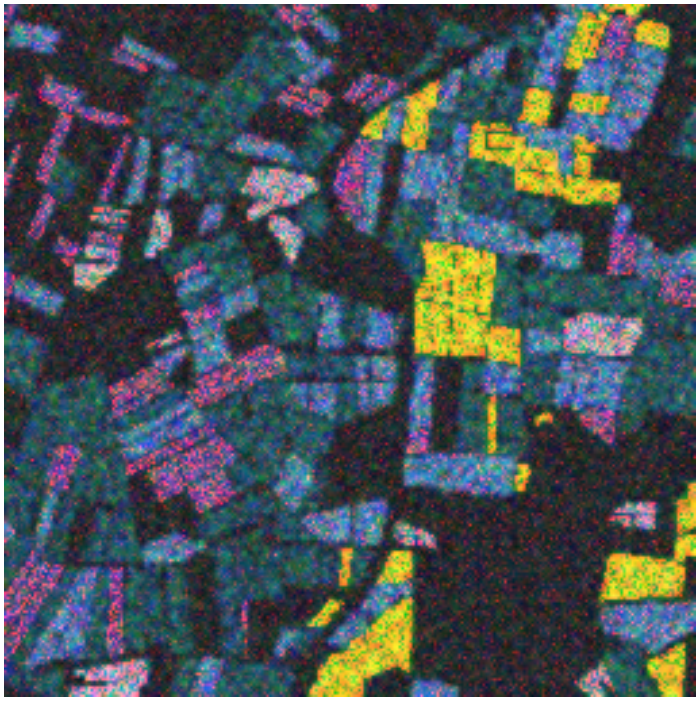

Figure 4.1. Simulated 3-look SAR data.

The shape parameter is most fundamentally related to the quality of the reconstruction, since it controls the smoothness of the segment boundaries, which affects the detail that can be retained in the reconstruction in the presence of speckle noise. This parameter is the one that has the most direct relationship with the properties of the original image. An image with considerable small-scale structure generally requires a large value of the shape parameter whereas a scene with little structure is best suited to a small value. In Figures 4.2 to 4.4 we illustrate the effect of the shape parameter on the result of segmenting Figure 4.1

From Figure 4.2 we observe that small details of scene structure are visible with a larger parameter value. However, there are also a number of small flecks in the reconstruction that are associated with speckle noise. This is known because the simulated images had uniform RCS within the segments. Thus this parameter value is too high. The result for the default shape parameter value in 4.3 has lost detail, such as the weak dashed lines visible in places in (a), but has also lost the spurious spots. As the value decreases, the reconstruction becomes progressively smoother with more detail averaged out. It is apparent that a value around the default value is probably "best" for this segmentation.

The other two parameters relate to the optimisation process and are less obviously related to the quality of the output. In fact the stopping parameter has little impact on the quality or speed of reconstruction, provided that the default values are used. The cooling parameter has comparatively little effect on the quality of the reconstruction but has a considerable impact on execution time.

For convenience, three predefined parameter sets are provided with `segment`: *fast* provides for a quick look at the segmentation; *default* is recommended for normal running; and *sensitive* provides suitable settings when the interest lies in small features.

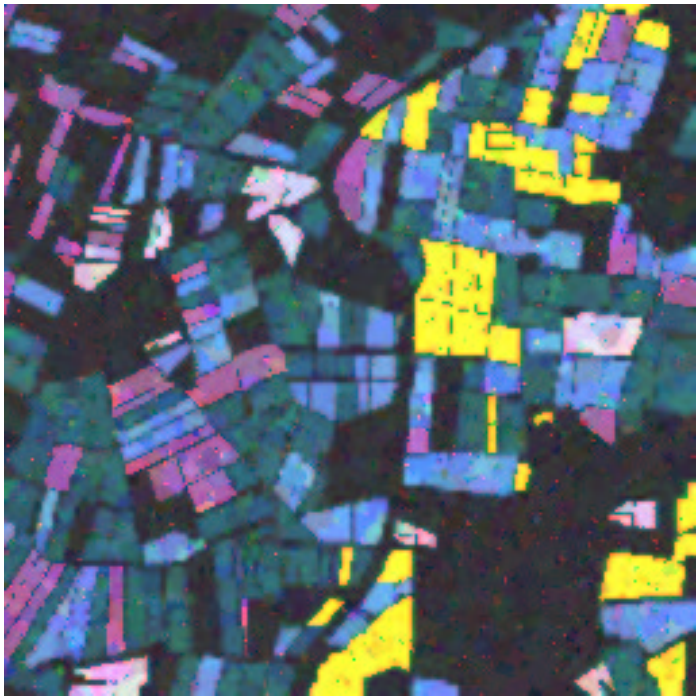

Figure 4.2. Segmentation with large shape parameter.

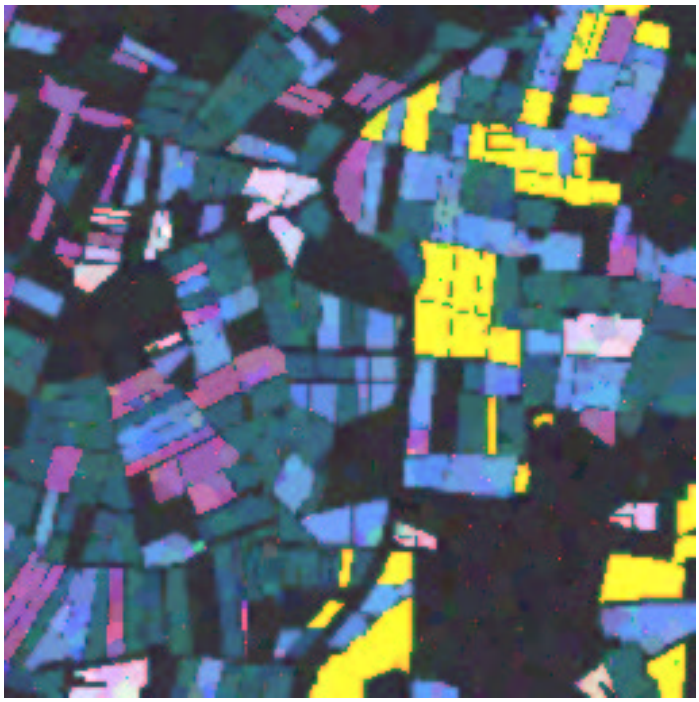

Figure 4.3. Segmentation with default shape parameter.

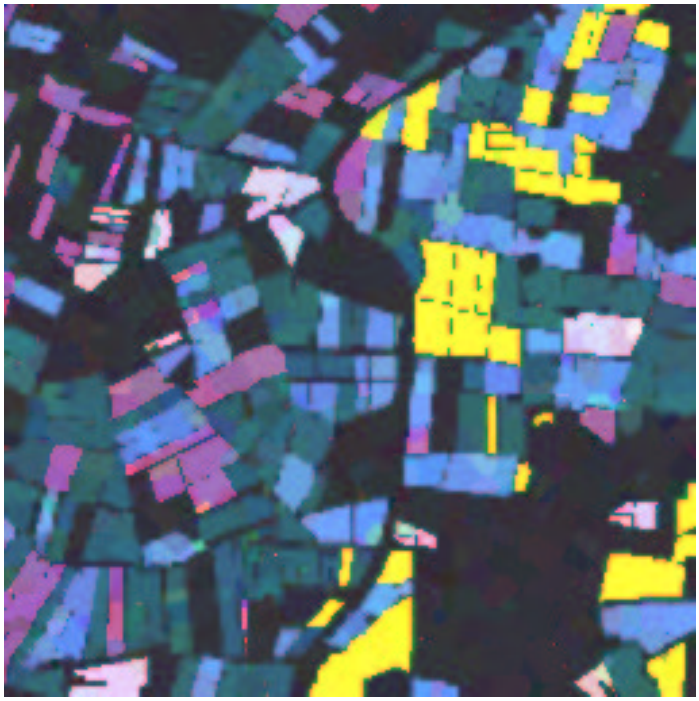

Figure 4.4. Segmentation with small shape parameter.

4.1.2 The `easyseg` routine

The `easyseg` routine is included to give an easy to use, quick, and smooth segmentation. It uses a call to `segment` followed by a call to `merge`; the parameters used have been preset to provide a fast but effective overall result. We recommend that you use `easyseg` first; if you find you need closer control of the process, call `segment` and (optionally) follow this with a call to `merge`.

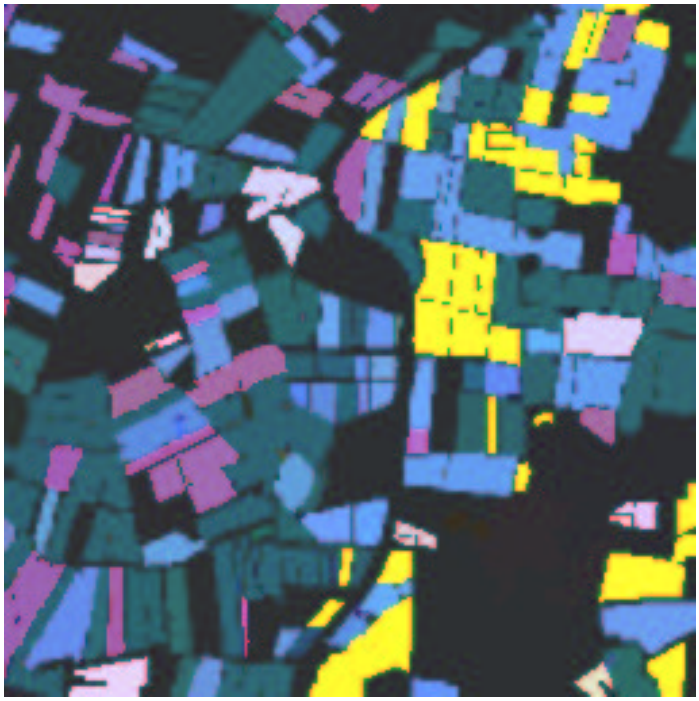

Figure 4.5. Merging the result from Figure 4.2.

4.1.3 The `merge` routine

For each pair of neighbouring regions, `merge` calculates the probability that the two regions are taken from the same underlying cross-section. If this exceeds the user specified probability threshold then the two regions are merged. This is repeated until there are no more regions suitable for merging.

From the original image and a region map of this image, `merge` will join any adjacent regions where the intensity does not significantly differ. The final number of regions can be controlled by the method used and associated cost:

- The probability threshold, `pfa` - smaller values of cost give fewer regions.
- The crinkle value, `crinkle` - same as `pfa` but also smooths the region boundaries.
- The absolute intensity value, `abs` - merges regions whose intensity differs by less than the entered cost value.

Figure 4.5 shows the result of merging the segmented image of figure 4.2. The less significant edges have been removed from the original cartoon image. Whilst the edges in the results produced by `segment` can, in theory, be of any statistical significance, `merge` provides a method to ensure that only highly significant edges remain.

The `merge` routine can process layered images. Full Python script and function syntaxes are given in section A.4.4.

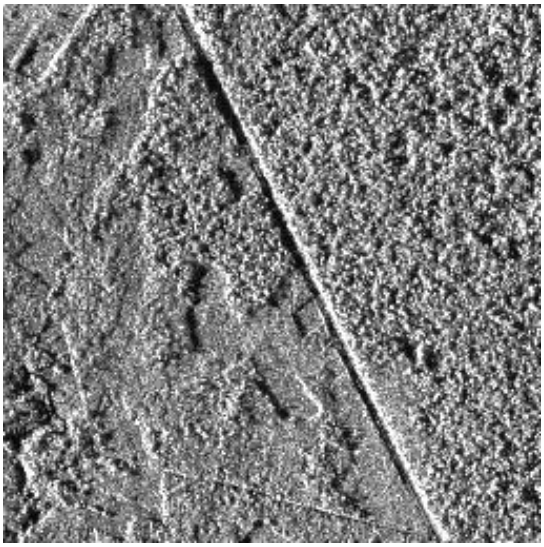

Figure 4.6(a)

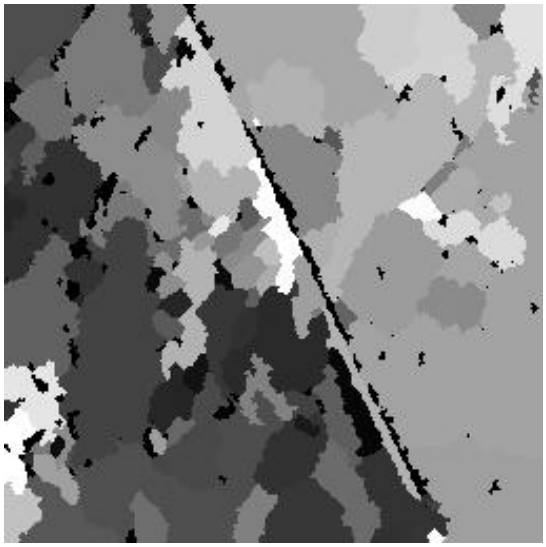

Figure 4.6(b)

Figure 4.6. (a) SAR image of rain forest. (b) Segmentation using `texseg`.

4.1.4 The `texseg` routine

Texture segmentation involves partitioning of an image into regions, each of which contains a single texture distinct from its neighbours. It works in an analogous fashion to segment, the difference being that instead of segmenting upon the measure of regional mean intensity, `texseg` segments upon the Blacknell-Tough measure:

$$\overline{I\log(I)}/\overline{I} - \log(\overline{I})$$

where I denotes the intensity image values. This measure accounts for variation of texture within the image. Since the measure is unreliable with small sample sizes and because the user should be looking to identify reasonably large regions with homogeneous texture, `texseg` starts with a default segmentation of regions of approximately 200 pixels in size. This value is user settable.

Figure 4.6 shows an example segmentation using `texseg`. Note that the routine has clearly differentiated between areas of virgin rain forest and clearings. This distinction would have been far less clear from using the `segment` routine. Note that the output images are also coloured according to the Blacknell-Tough measure.

Joint segmentation is performed when there is more than one image layer. Full Python script and function syntaxes are given in section A.4.3.

4.1.5 The `colseg` routine

`colseg` is a routine that will process a segmented image to colour regions according to one of a number of measures. This is useful, both visually and for further processing purposes, since it allows variation in the scene to be highlighted according to different properties.

Figure 4.7 shows the result of colouring the segmentation of figure 4.6 by two different methods.

The `colseg` algorithm can be performed upon images with more than one image layer. Full Python script and function syntaxes are given in section A.4.5.

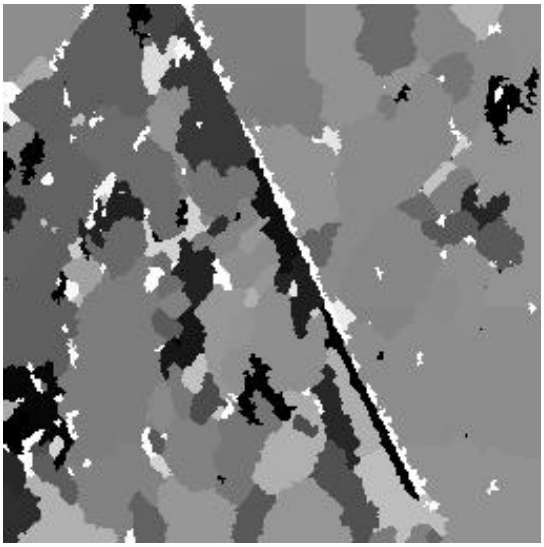

Figure 4.7(a)

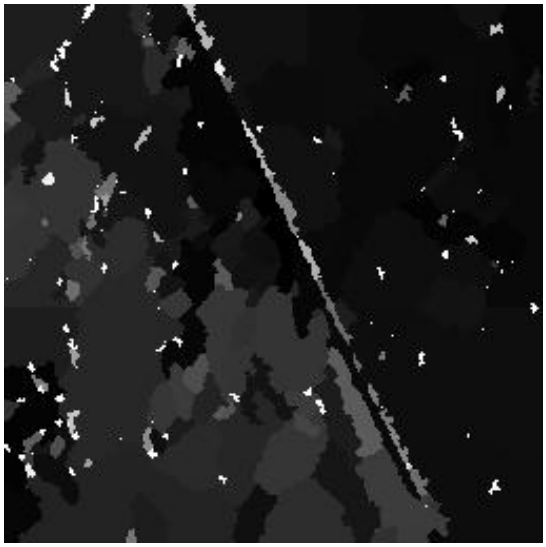

Figure 4.7(b)

Figure 4.7. Result of running `colseg` on the image of Figure 4.6(b) with (a) colouring by mean intensity. Note that the difference between clearing and virgin rain forest is much less pronounced than when the regions are coloured using the Blacknell-Tough measure as in Figure 4.6(b). (b) colouring by regional Gamma order parameter.

4.1.6 The `defleck` routine

It is often the case after a segmentation, or a segmentation and merge, that you end up with unwanted single pixel flecks across the image. By running the Defleck routine, you can get rid of these flecks without losing other information.

4.2 Classification

A classification algorithm is similar to a segmentation algorithm in that it produces a cartoon image in which each pixel is assigned to one of a set of predefined classes. The number of regions in a segmentation may be quite high, but these segments may usually be classified into a small number of categories: for example fields/roads/hedges; forest/clearings; potatoes/wheat/ploughed/grass. For many purposes the *classification* is more useful than the *segmentation*.

4.2.1 The `classify` routine

`classify` performs Maximum Likelihood classification of intensity images. Maximum likelihood classification is based upon the assumption that there exist statistical models describing the distribution of the classes in the attribute space. Given these models, the class of a new object is determined by calculating which of the models is more likely to describe that object. In other words, the model with maximum likelihood is selected.

Starting with a segmentation of the original image, which has the correct statistical model for the data and a plausible boundary model, `classify` will allocate each region to one of a number of classes. The output is a single image with the regions labelled with the class number (ie 0,1,2...)

InfoPACK supports both supervised and unsupervised classification:

Unsupervised Classification In unsupervised classification, the class values will be derived from uniform spacing of the original data. The number of classes is determined by the user setting the parameter `num_classes` in the routine `make_classes`. This generates a means file which can then be passed to the `classify` routine.

Supervised Classification With supervised classification, *a priori* class means or *a priori* class means along with class order image files are input to the routine. These files can be derived via:

- Entering the values in a text file and using the `ascii2nc.py` utility. The values could be selected by using `histogram.py` to study the distribution of the data and pick out peak values.
- By running the `mask_stats` or `classstats` routines on a set of training masks. See the `images` directory for an example of this.

In any case, the `means.nc` and `orders.nc` files can be edited further to provide iterative supervised classification.

Figure 4.8 shows one of a sequence of images of the Indonesian rain forest and the resultant image from a joint segmentation process. Using this segmentation, the images can then be classified as shown in figure 4.9. The reason for using the sequence of images for classification is that the information is in the temporal change. Joint segmentation provides better definition of boundaries. The low resolution and speckle associated with just a single image would give rise to a much poorer result.

A further example of supervised classification is given in the `images` directory. Full Python script and function syntaxes are given in section A.4.7.

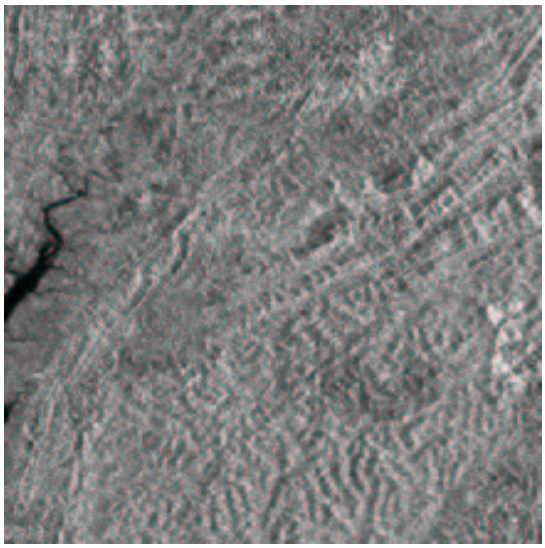

Figure 4.8(a)

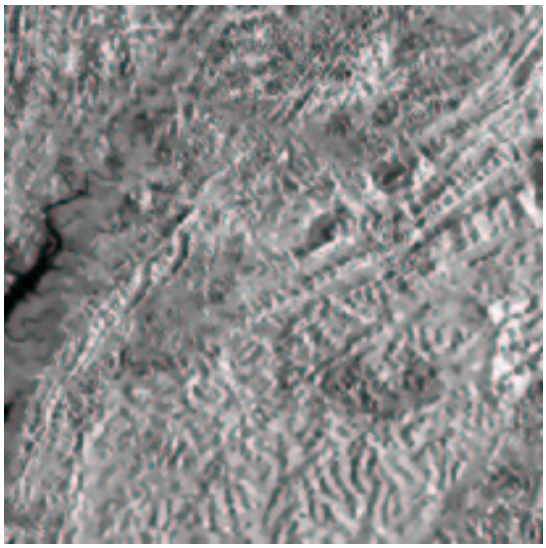

Figure 4.8(b)

Figure 4.8. (a) Original SAR image of Indonesian rain forest. (b) Result of joint segmentation.

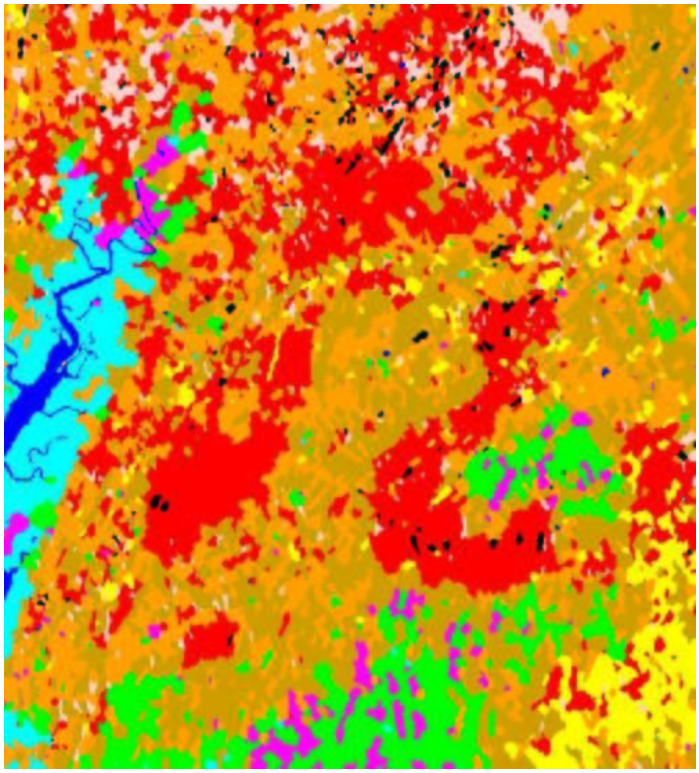

Figure 4.9. Result of using `classify` for the original SAR image of Figure 4.8(a).

4.2.2 The `mask_stats` routine

The `mask_stats` routine calculates the mean and order parameters, necessary for supervised classification, from the masked regions of an image. Each layer of the mask image should indicate the training data for each class. The resultant file is a single layered real valued image with height equal to the number of layers and width equal to the number of classes.

There is an example of a temporal sequence of SAR images, a sequence of associated mask images and detailed instructions on how to process the images using `mask_stats` and `classify` in the `images` directory. Full Python script and function syntaxes are given in section B.5.5.

4.3 Area Change Detection

These routines detect large–scale changes between two or more images; *large–scale* implies that they are not tailored to detecting differences in point targets between images. The images to be scanned for changes must be provided as layers in a single multi–layer image (if your images are separate use the `tools/Layer Operations` option in the GUI to merge them into a single multi-layered image). The routines use the segmentation of each layer to carry out their analysis, and this must be provided too. In most cases the "joint segmentation" produced by segmenting the multi-layer image is what will be conveniently used; but if (for example) you are interested in changes in field boundaries, then segment each layer separately and merge the individual segmentations into a multi-layer image before calling the change detection routine. You would use Change_detection if you have an unknown sequence of several changes. The algorithm basically detects whether the mean remains constant, or varies throughout the sequence. Change position detection introduces further prior knowledge, namely that there is one change between constant levels. Since the prior knowledge is stronger, the result is better. If, however, you try to use it where there are several changes, such as in agricultural monitoring, then the results will not be of use.

4.3.1 The `change_detection` routine

`change_detection` looks for significant change over a series of original speckled images. The routine detects changes across the whole series of provided image layers - the ordering of the layers in the multiple layer image file is not important. The output is a single layer image specifying where any significant changes were found over the whole sequence. If you are more interested in the individual changes found between adjacent pairs of layers in the input image, use routine `change_detection_position`.

Figure 4.10 shows an example sequence of four multi-temporal images. The changes detected by `change_detection` (at the default level of significance) are shown in figure 4.11(a).

Full Python script and function syntaxes are given in section A.4.8.

4.3.2 The `change_detection_position` routine

This routine detects changes between adjacent pairs of layers in the provided image - the ordering of the layers in the multiple layer image file is therefore crucial. The output is a multiple layer image with number of layers equal to the number of adjacent pairs in the sequence of layers in the original image (i.e. number of layers minus 1). Each of these layers indicates where significant changes were found between the image pair. This routine is useful for example when monitoring the growth of crops over the seasons. Figure 4.11 shows an example application of this routine with respect to the sequence of images of figure 4.10.

If you are interested solely in whether there are changes over the whole sequence of layers in the input image, use routine `change_detection`.

Full Python script and function syntaxes are given in section A.4.9.

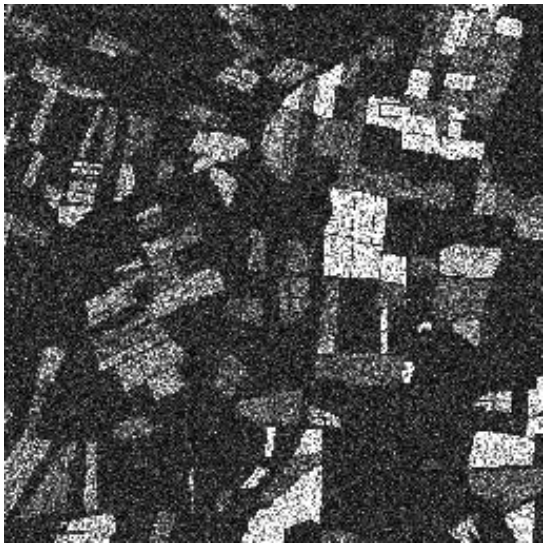

Figure 4.10(a)

Figure 4.10(b)

Figure 4.10(c)

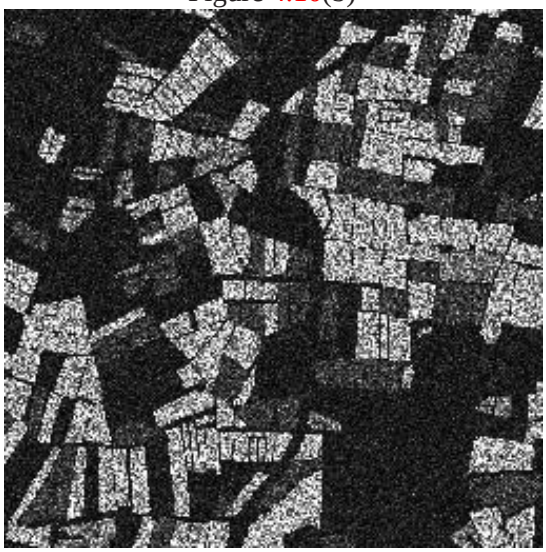

Figure 4.10(d)

Figure 4.10. Multi-temporal sequence of SAR images.

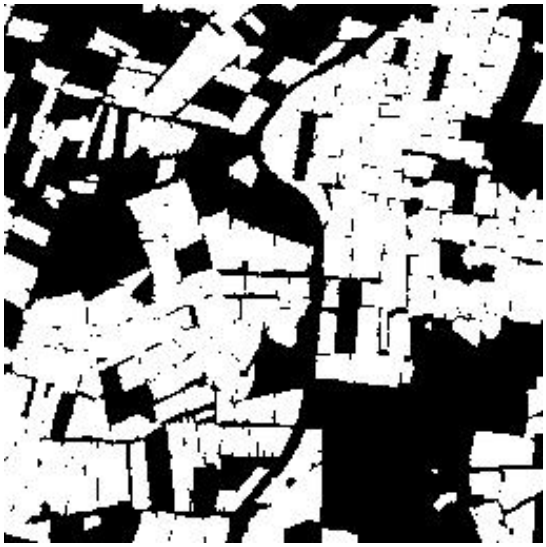

Figure 4.11(a)

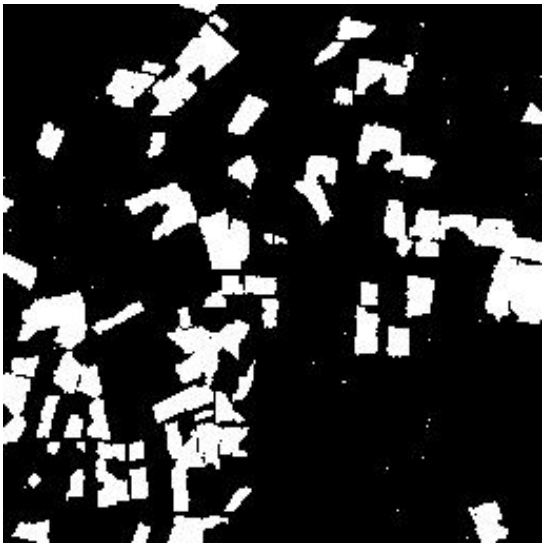

Figure 4.11(b)

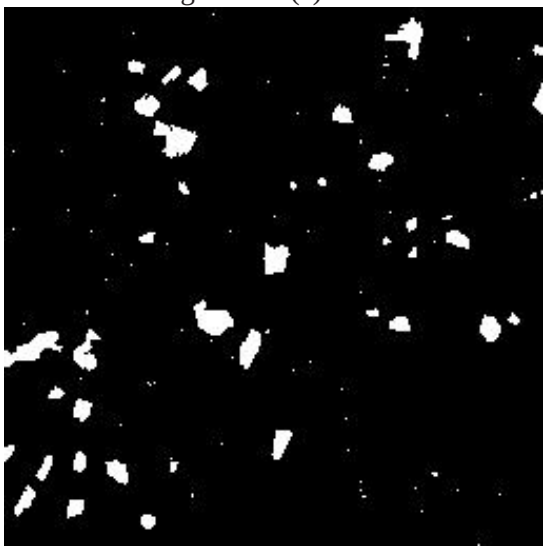

Figure 4.11(c)

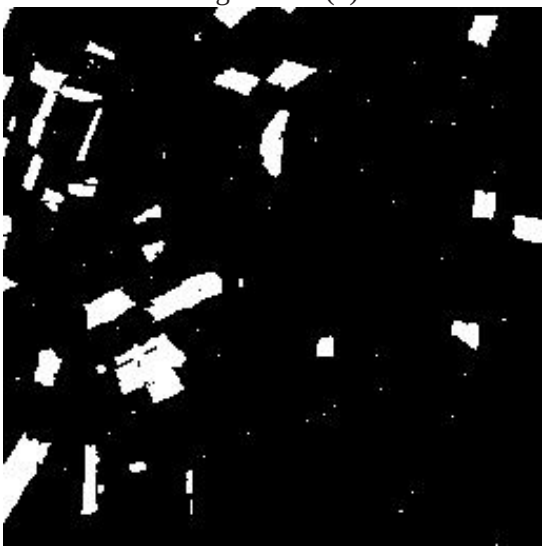

Figure 4.11(d)

Figure 4.11. Detecting changes in the sequence of images of Figure 4.10.
(a) Result of `change_detection`.
(b) Result of `change_detection_position` between Figures 4.10(a) and 4.10(b).
(c) Result of `change_detection_position` between Figures 4.10(b) and 4.10(c).
(d) Result of `change_detection_position` between Figures 4.10(c) and 4.10(d).

4.4 Detecting Edges

Detecting edges and region boundaries is a fundamental requirement for automated image interpretation. A segmentation algorithm divides up an image into regions. An edge detector finds lines which separate areas with different characteristics: these edges do not necessarily join up to form boundaries of closed regions.

An edge-map is a binary image in which each pixel indicates whether or not an edge is present. It can be produced directly from the raw data by an edge detection algorithm, or (usually much more reliably) by identifying the region boundaries in a segmentation.

4.4.1 The `scanedge` routine

The `scanedge` function detects edges in a raw SAR image via a combination of a scanning window and a scanning edge. The algorithm used has been shown theoretically [2] to give optimal detection of edges in a raw SAR image, for the window size specified. The sensitivity of the detector can be altered by setting the probability of false alarm (*pfa*) — the probability of finding edges where none exist. Smaller *pfa*s lead to fewer edges being found.

The `scanedge` algorithm can be performed upon images with more than one image layer. `scanedge` can also be accessed via the `filter` function: use `method=''scan''`. Full Python script and function syntaxes for `scanedge` are given in section A.4.11.

Figure 4.12 shows a raw DLR ESAR amplitude image, and edge-maps produced from it for several combinations of filter size and pfa. It is clear that the dominating parameter is the filter size: we recommend using a size of 7 by 7.

4.4.2 The `edge` routine

Alternatively, by using the `edge` utility, you can produce an edge-map of the region boundaries from a segmentation. An edge-map overlaid on its corresponding cartoon may show up region boundaries you would otherwise overlook: two adjacent regions may have similar intensities.

Do *not* use `scanedge` to convert a cartoon image, the output of a segmentation function, into an edge-map. Use `edge` to perform this task.

An example of the use of `edge` is shown in Figure 4.12(f). The `edge` algorithm can be performed upon images with more than one image layer. Full Python script and function syntaxes for `edge` are given in section B.2.5.

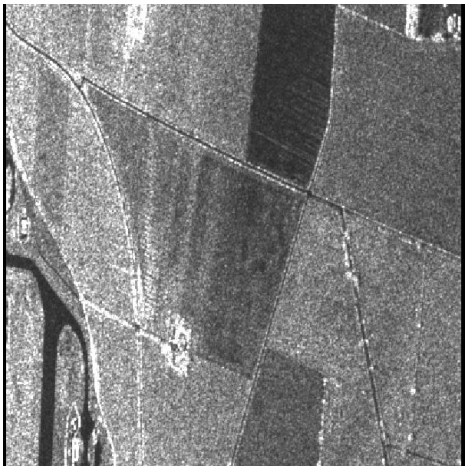

Figure 4.12(a)

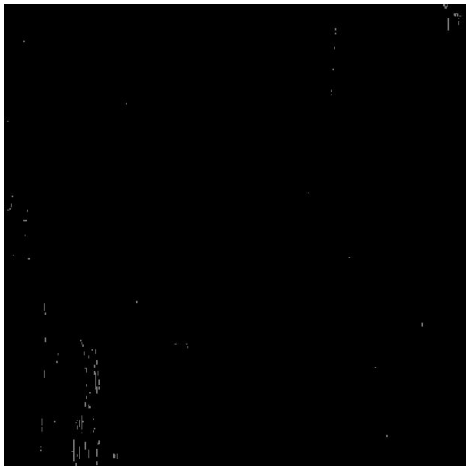

Figure 4.12(b)

Figure 4.12(c)

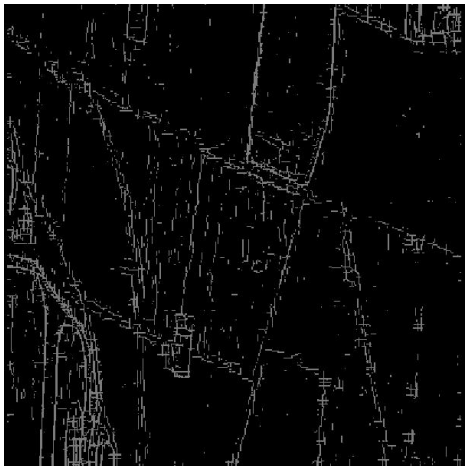

Figure 4.12(d)

Figure 4.12(e)

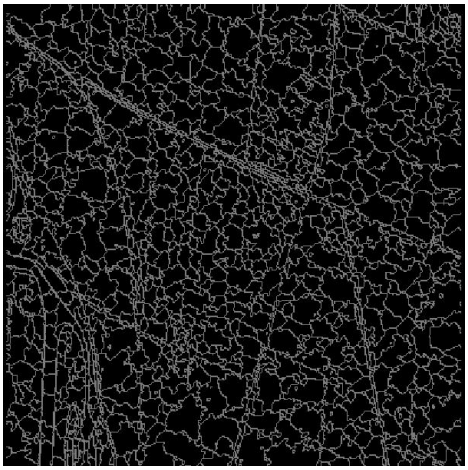

Figure 4.12(f)

Figure 4.12. (a) Raw ESAR Image (b) `scanedge` output with filter size 3 by 3 and pfa 10^{-5} (c) 7 by 7 (d) 10 by 10 (e) 7 by 7 with pfa 10^{-7} (f) Edges found from the segmented image using `edge`

4.5 Speckle Reduction

Normally it is not necessary to call a specific speckle-reduction routine first (or indeed at all). However, when required `despeckle` will provide a despeckled version of a given image without unnecessarily losing detail.

4.5.1 The `despeckle` routine

`despeckle` approximates a single or multiple image with a locally-constant (i.e. locally-flat) model of the surface cross-section. It fits this model using a variant of the simulated annealing optimisation method. The noise model used in the procedure is determined by the looks measure for the image. A higher value of looks measure implies that the data are less influenced by noise and are, therefore, more reliable estimates of the underlying surface.

The *InfoPACK* routines all assume the input image has uncorrelated speckle. If your image is oversampled so that it includes correlations then use the `shrink` utility to remove them; see Section 6.5 for a discussion of how to do this optimally.

The `despeckle` algorithm can be performed upon images with more than one image layer. Full Python script and function syntaxes for `despeckle` are given in section A.4.10.

The effects of the parameters of the speckle-reduction functions are described below.

The number of iterations

`despeckle` is iterative. The longer it runs, the smoother the resulting surface. Figure 4.13(c) shows the effect of smoothing a section of a DLR ESAR image for 10, 50 (the default value), and 250 iterations using `despeckle` with the looks measure set to 1 and other parameters set to the default values.

The looks measure parameter

Figure 4.14 shows the result of increasing the looks measure used by the routine `despeckle`. Fifty iterations were used so compare it with Figure 4.13. When the looks measure parameter is increased, the algorithm interprets the input data as being less affected by noise. This means that `despeckle` will be more likely to interpret fluctuations caused by speckle as genuine jumps in the cross-section. The output is less smooth as a result. We recommend that normally you let the routine estimate the looks measure for itself.

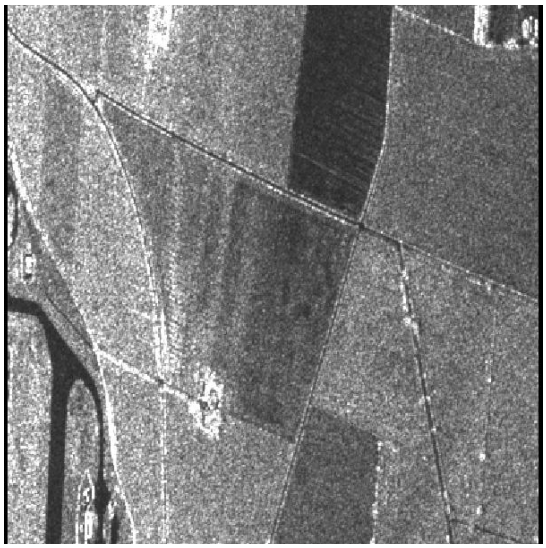

Figure 4.13(a)

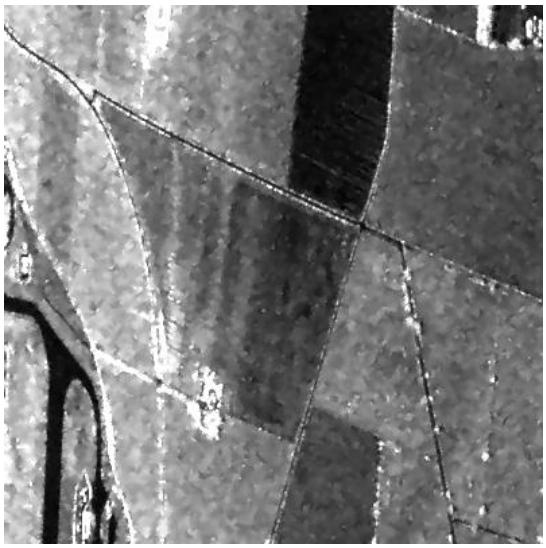

Figure 4.13(b)

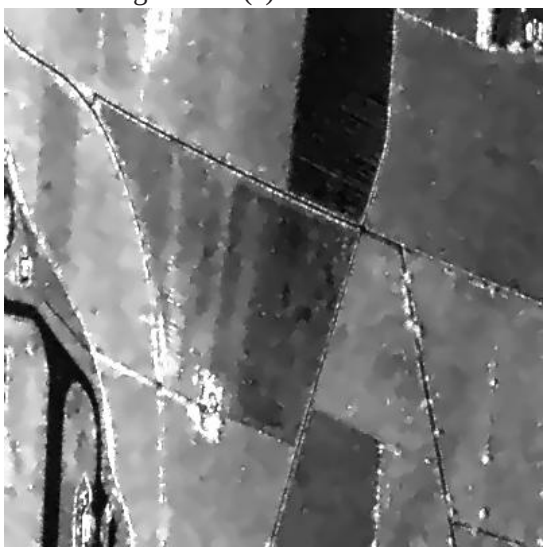

Figure 4.13(c)

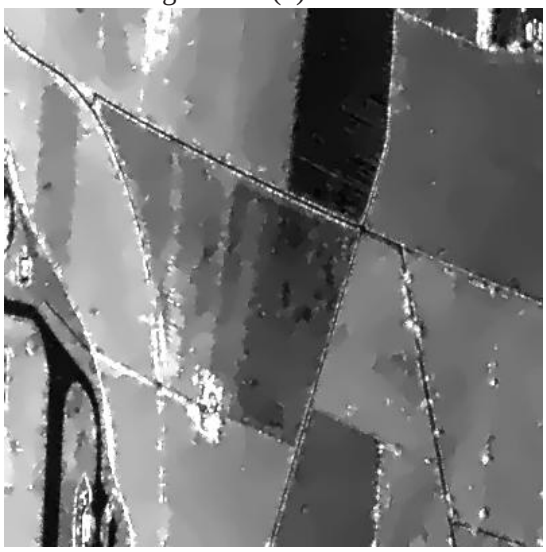

Figure 4.13(d)

Figure 4.13. (a) Unprocessed SAR image (b) despeckled with looks measure = 1 and 10 iterations (c) 50 iterations (d) 250 iterations

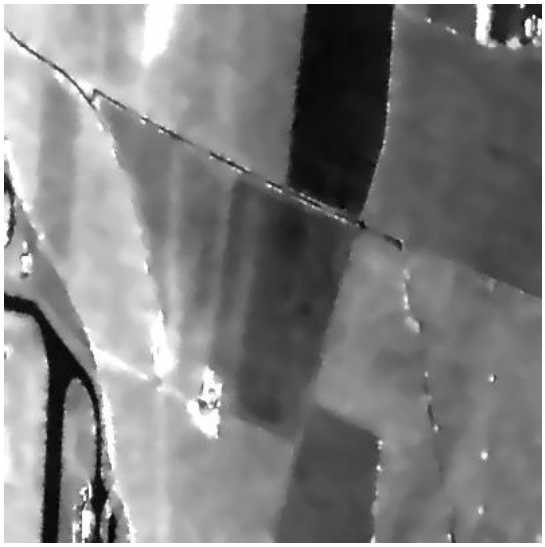

Figure 4.14(a)

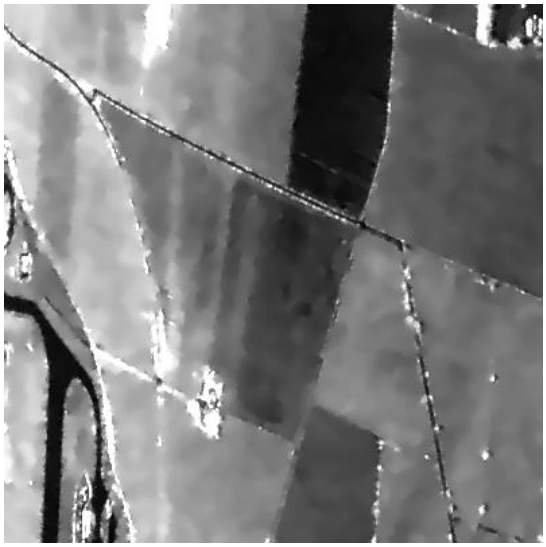

Figure 4.14(b)

Figure 4.14. Despeckling the image of figure 4.13(a) with 50 iterations and (a) looks measure = 2 (b) looks measure = 4

Figure 4.15(a)

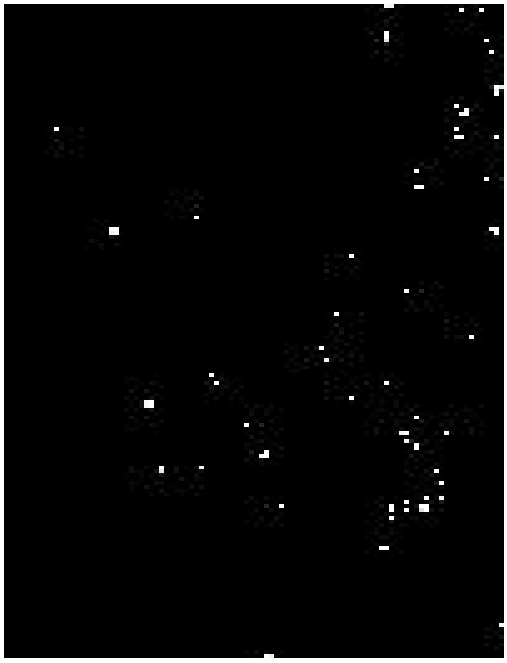

Figure 4.15(b)

Figure 4.15. (a) Original image containing potential point objects. (b) Image processed by `cfar` to detect point objects.

4.6 Point Object Detection

Point objects are bright areas of an image only a few pixels in size. They correspond to highly reflective scatterers on the ground — often metal objects such as vehicles or pylons. Because they are small and isolated, they can be difficult to distinguish from the fluctuations due to speckle.

4.6.1 The `cfar` routine

The `cfar` algorithm locates bright areas in the image which are not just the effect of speckle, and checks to see whether they are small and isolated.

The CFAR algorithm works by calculating the likelihood of the central pixel in a scanning window coming from the background distribution (as represented by the rest of the window). The classification of the central pixel as a point object depends upon the comparison of this likelihood with the threshold calculated from the probability of false alarm (`pfa`). This probability can be set by the user. A lower value of probability of false alarm will mean less point objects will be detected. Since the result is determined by the accuracy of the statistical estimates, it is advisable to have a reasonably large data sample, i.e. window size. We suggest using a 7x7 window. Figure 4.15(b) shows the point objects detected in figure 4.15(a) through the use of `cfar`.

The `cfar` algorithm can be performed upon images with more than one image layer. `cfar` can also be accessed via the `filter` function: use `method=''cfar''`. Full Python script and function syntaxes for `cfar` are given in section A.4.11.

4.7 References

For papers to download, other references and links please visit our website
`http://www.nasoftware.co.uk`

Bibliography

[1] Speckle reduction
I. McConnell, R. G. White, C. J. Oliver and R. Cook, *Radar cross-section estimation of SAR images*, Proc. EUROPTO (Paris, 1995). SPIE Vol 2584: Synthetic Aperture Radar and Passive Microwave Sensing.

[2] Edge detection
C. J. Oliver, D Blacknell, I. McConnell and R. G. White, *Optimum edge detection in SAR*, Proc. EUROPTO (Paris, 1995). SPIE Vol 2584: Synthetic Aperture Radar and Passive Microwave Sensing.

[3] Segmentation (`texseg`)
C. J. Oliver, I. McConnell and D. Stewart, *Optimum texture segmentation of SAR clutter*, Proc. EUSAR Conference (Königswinter, 1996).

[4] Segmentation (`segment`)
D. Stewart, D. Blacknell, A. Blake, R. Cook and C. J. Oliver, *An optimal approach to SAR image segmentation and classification*, IEE Proc. Radar, Sonar, Nav. 2000

Chapter 5

Using the InfoPACK GUI

5.1 Introduction

The *InfoPACK* GUI provides easy access to the *InfoPACK* Image Understanding Library by means of a set of menus and dialog boxes. Routines are run within the main Python Shell Window. Scripts, Python commands and *InfoPACK* functions can also be run from within the shell window. Also supplied is the Open Source Python Integrated Development Environment, "IDLE", which provides a language sensitive editor and debugger for rapid development of Python scripts and complete processing chains.

5.2 Starting The *InfoPACK* GUI

5.2.1 Windows

Navigate to the the *InfoPACK* entry of the Programs folder of your Start menu and click on it.

5.2.2 Unix/Linux

On Unix systems, just type `InfoPACK` at a command prompt.

5.3 The Main *InfoPACK* Menus

The easiest way to access the *InfoPACK* library is via the menus in the main *InfoPACK* GUI window. Clicking on menus will generally bring up a dialog box from which files are selected, parameters are chosen and the routine is run. All menu items call the appropriate *InfoPACK* script. All *InfoPACK* scripts work file-to-file, to enable the processing of arbitrarily large images within memory. As such it is always necessary to provide an output file name in the appropriate dialog box. All menu commands are executed from within the Python shell. When a command is processed you will see in the shell window the script you have called and the parameters you are using. For ease of use, all recently accessed image file names are stored which can be accessed from a "recent files" button in dialog boxes and the variable "files" in the shell window.

The main toolbar on the GUI screen includes the following menus:

File (see section 5.3.1) with menu items:

```
Open Image
Import Image
Export Image
Run Study
New Script
Open Script
New Console
Exit
```

Tools (see section 5.3.2) with menu items:

```
Create New Image
Layer Operations
Extract Sub-Image
Resize Image
Histogram Image
Image Statistics
Image Properties
Clip Image
Filter
Despeckle
Threshold
Image Arithmetic
Stripmap Options
```

Segmentation see section 5.3.3

```
Segment
Texture Segment
Merge
Colseg
Defleck
Edge
```

Classification see section 5.3.4

```
Classify
Generate Classes
Colour Classes
Class Statistics
Classmap to Layers
Layers to Classmap
Combine Matrix
Split Matrix
```

Change Detect see section 5.3.5

`Change Detection`
`Change Position Detect`

Help see section 5.3.6

`Create Licence File` Enter your licence details to activate the InfoPACK routines.

`InfoPack` This documentation, including documentation for all InfoLIB routines. Most users will need only this manual.

`Python` For details of the base Python language.

`Numeric Python` The manual for the Numeric Python extensions (used by *InfoPACK*).

`Command List` This pops up a window listing the python scripts that come with *InfoPACK*.

`About` Gives the version number for this release.

5.3.1 The File Menu

This provides access to the following facilities:

`Open Image`

Open an image file in a format supported directly by the OpenEV Image Viewer (see Section 5.4).

`Import Image`

Convert an image into the *InfoPACK* standard `.nc` format (see chapter 3). A wide range of standard image formats is supported.

`Export Image`

Convert a `.nc` image file to an alternative format. Full conversion to the following formats is provided:

- GeoTiff (*.tif)
- Tiff Bitmap (*.tif)
- ERDAS Imagine (*.img)
- Atlantis MFF (*.hdr)

Other export formats, including postscript and `.png`, are made available from the `OpenEV` viewer: Use the Print option and print to file.

`Run Study`

This will prompt for the case study file to be opened, and then show the output of the study in the console. Under Linux, the case study should end with `.lin` while under Windows the case study should end with `.bat`.

`New Script`

This opens a blank `IDLE` shell window. From the file menu, selecting "New Window" brings up an empty `IDLE` editor window, ready for you to type in a Python script: a text file containing a list of *InfoPACK* (or other Python) commands to be executed. Python script files are generally give the suffix `.py`. For details of the editor features, see section 5.7.

`Open Script`

This will prompt for the script file to be opened, and then open it in a new `IDLE` editor window. An `IDLE` shell window is also opened for the script output when run from the editor window. For details of the editor features, see section 5.7.

`New Console`

This will open a new *InfoPACK* shell window.

`Exit`

Exit *InfoPACK*.

5.3.2 The Tools Menu

`Create New Image`

Create an image with artifical speckle generated from a specified distribution. Useful when creating test images. Full function specifications are given in Appendix B.

`Layer Operations`

This provides access to operations for generating a 3-D (layered) image from a set of single layer images, and for extracting individual layers from a 3-D image. Two options are provided:

- `Split Layers`: Splits a 3-D image into a numbered set of the individual layers.
- `Combine Layers`: Combine a set of individual files into a single 3-D image.

Full function specifications are given in Appendix B.

`Extract Sub-Image`

Extract a subimage from a given image. You will be prompted for the subimage size (width and height), and for the w (width) and h (height) offsets. Full function specification is given in Appendix B.

`Resize Image`

Resize an existing image. You will be prompted for the new image size (width and height), and for the method of interpolation. Full function specification is given in Appendix B.

`Histogram Image`

Produce a histogram showing the distribution of amplitudes/intensities of an image. For multi–layered images, the histogram for individual layers can be viewed as well as for the whole image data. This utility can be useful when planning to carry out supervised classification of an image. Full function specification is given in Appendix B.

`Image Statistics`

Provides a set of statistics on the specified image:

- Image size and datatype
- Minimum and maximum pixel values
- Mean and standard deviation of pixel values
- The looks measure and average local looks measure
- Order (of equivalent gamma distribution) and average local order

Full function specification is given in Appendix B.

`Image Properties`

Provides information on the selected image, including:

- Size
- Coordinates of origin, image corners and image centre
- Pixel Datatype for each layer (“Band”)

`Clip Image`

Allows you to clip a specified image with specified minimum and maximum values (default values of 0 and 100 respectively). Any pixels in the image with values less than the specified minimum will be replaced by the minimum; any with values greater than the maximum will be replaced by the maximum.

Full function specification is given in Appendix B.

`Filter`

This provides access to a range of low level filters found in traditional image processing packages. Filters perform a variety of operations (such as averaging, smoothing or edge detection) using a fixed window size.

See section A.4.11 for details of the routine parameters, None of the high level routines in InfoLIB use these filters; they are provided primarily as a convenience.

`Despeckle`

Despeckle an image. See section A.4.10 for details of the routine parameters, and section 4.5.1 for a longer discussion.

Threshold

Produce a 0/1 "mask" image labelling pixels above/below a given threshold. Image pixels that lie above the threshold are set to the true value, otherwise to the false value. See section B.2.29 for details of the routine parameters.

Image Arithmetic

Two options are provided:

- **Single image functions:**
  Provides access to the following arithmetic operations on images, carried out on each pixel:
  - Scalar Add: image2 = image1 + c
  - Scalar Multiply: image2 = image1*c
  - Scalar Divide: image2 = image1/c
  - log10: image2 = $\log_{10}$(image1)
  - log e: image2 = $\log_e$(image1)
  - square root: image2 = $\sqrt{\text{image1}}$
  - square: image2 = image1*image1
- **Double Image Functions:** Provides access to the following operations, carried out on each corresponding pair of pixels in two specified images of the same size:
  `image3 = image1 op image2, op = +,-,*,/`

Full function specifications are given in Appendix B.

Stripmap Options

Options are provided to set the default `tilesize` and `overlap` values. These values have no effect with some evaluation versions of *InfoPACK*.
Full specifications are given in Appendix A.2.

5.3.3 The Segmentation Menu

Easyseg

Easyseg provides a fast and easy to use segmentation routine recommended for exploratory or routine use. It carries out an initial segmentation followed by a call to the `merge` routine. If you neeed closer control of the segmentation process, call `segment` and (optionally) `merge` separately.

See section A.4.2 for details of the routine parameters, and section 4.1.2 for a longer discussion.

Segment - Default Settings

Segment a given image file with one of three predefined settings "default", "fast" or "sensitive". See section A.4.1 for details of the routine parameters, and section 4.1 for a longer discussion.

`Segment - Custom Setting`

Segment a given image file with full control over the options. The default settings are shown and can be altered. See section A.4.1 for details of the routine parameters, and section 4.1 for a longer discussion.

`Texture Segment`

Segment a given image file using texture instead of intensity information in the image. The `default settings` allows you to choose from one of three predefined settings "default", "fast" or "sensitive". You will be prompted for the segmentation parameters or select the `custom settings` to view and change the default values. See section A.4.3 for details of the routine parameters, and section 4.1.4 for a longer discussion.

`Merge`

Merge removes weak edges from a segmentation. It does this by merging regions that are not significantly different in intensity at the specified level of significance. `Merge` requires the segmented image, the original image, and a significance level. See section A.4.4 for details of the routine parameters, and section 4.1.3 for a longer discussion.

`Colseg`

Given a segmentation image and the original imge, colour the segmentation (ie floodfill each segment) with a value computed from any of a set of measures. See section A.4.5 for details of the routine parameters, and section 4.1.5 for a longer discussion.

`Defleck`

Use Defleck to remove small spots that sometimes occur after using `Segment` or `Merge`. See section A.4.6 for details of the routine parameters, and section 4.1.6 for a longer discussion.

`Edge`

Produce an edge map showing the edges of a segmented image. See section B.2.5 for details of the routine parameters, and section 4.4.2 for a longer discussion.

5.3.4 The Classify Menu

`Classification`

Classify an image into a given number of classes. `Classify` requires a segmented image, the original image and a file with the class means and an optional file with the order parameters for supervised classification. See section A.4.7 for details of the routine parameters, and section 4.2 for a longer discussion.

Generate Classes

There are four routines under the heading Generate Classes. These are:

- Make Classes: If no apriori classes are available, this routine generates a set of equally spaced class means from the input image.
- Mask Statistics: This routine uses training masks to calculate the mean and order statistics files needed for classification.
- Mask Polstats: Classifying polarimetric data needs to generate the class covariances which this routines does with the help of some region masks.
- Class Statistics This routine generates a mean and orders file from a classification. For each class labelled 1,2,3,..., the mean and order parameter is calculated from the input image.

Colour Classes

Use this routine to give false colour to your classified image allowing you to view all classes together in one image, instead of the maximum of four allowed with OpenEV.

Classmap2Layers

A classified image has regions labelled 1,2,3,.... This routine splits the classification into layers (or masks) so that the first layer is a binary (0/1) mask of those region in class 1. The second layer is a mask of class 2 and so on.

Layers2Classmap

This routine converts a set of binary (0/1) masks into a classification. The pixels in the first layer are converted to class 1. The pixels in layer 2 become class 2 and so on.

Display Class Means

Given a classification means files, this routine prints out the class means in an easy to understand format.

Combine Matrix

The purpose of this routine is to combine a set of classification "means" files into a single file. Use this routine on a set of means files as you would use Layer Combine in section B.3.8 on images.

Split Matrix

If a classification means file is generated from a multi-temporal image, this routine splits the means file into seperate files as Layer Split would split the original multi-temporal image into layers.

5.3.5 The Change Detect Menu

`Change Detection`

Detect changes between layers of a multiple-layer image: for example, a multi–temporal sequence. The routines accessed look for large scale changes between layers of the specified multi–layer image; you need to supply also the segmented image.

Two options are provided:

`Change Detect`: produces a single layer output image denoting areas in which changes were found between any of the images. See section A.4.8 for details of the routine parameters, and section 4.3.1 for a longer discussion.

`Change Position Detect`: detects changes between adjacent layers. The output is an image with one less layer than the input image; each output layer contains the changes found between one adjacent pair of layers in the input image. This option is useful for example when monitoring the growth of crops over the seasons. See section A.4.9 for details of the routine parameters, and section 4.3.2 for a longer discussion.

5.3.6 The Help Menu

`Create Licence File`

Use this to input your licence details. Without these you will not be able to run any of the `InfoPACK` routines.

`InfoPACK HTML`

Gives you access to the InfoPACK HTML documentation.

`InfoPACK PDF`

Gives you access to the InfoPACK PDF documentation.

`Python`

Gives you access to the Python documentation.

`Numeric Python`

Gives you access to the Numeric Python documentation.

`Command List`

This displays a window listing the python scripts available with *InfoPACK*.

`About`

Gives the version number of *InfoPACK*.

5.4 The OpenEV Image Viewer

InfoPACK provides the OpenEV image viewer to view raw or processed images. This open source viewer is well suited to handling large images of a wide variety of formats. It also takes account of geo-referencing information when provided and can produce overlay images (for example map data overlaid upon co-registered SAR data). Documentation for OpenEV is available from OpenEV's `Help` menu.

5.4.1 Classification Images

In order to save disk space, *InfoPACK* classification images are stored as byte data. OpenEV, however, interprets byte images as greyscale bitmaps and so the classes 1,2,3... get displayed as black.

There are two workarounds to this problem:

- Use the colour classes routine (`colclass`) to convert the classes to red, green, blue and so on.
- Use the image equalisation buttons (See Figure 5.1) in OpenEV to remap the class values to a more useful greyscale.

Figure 5.1. OpenEV image equalisation buttons.

5.5 Using the *InfoPACK* Shell Window

The Shell Window provides access to an interactive mode. This window will open as part of the main screen when you fire up the *InfoPACK* IDE, and packages needed to provide access to *InfoPACK* facilities are automatically loaded. You can type any python statement into this window; but most users will need only those facilities provided specifically by *InfoPACK*, and documented here. Full *InfoPACK* Python function specifications are given in Appendices A and B.

As with the Python direct interactive mode (using Python from a command window), you type a Python statement into the shell window at the >>> prompt and then hit the Enter key to send it to the Python interpreter.

InfoPACK scripts can be executed directly from within the shell. For example, to segment the image "myimage.nc" to the output file "mysegment.nc" with default parameter settings, you would type:

```
>>> run("segment.py myimage.nc mysegment.nc")
```

InfoPACK functions can similarly be called. For example, to load "myimage.nc" into memory, segment it, then save the result as "mysegment.nc" you would type:

```
>>> array_in = numload("myimage.nc")
>>> array_seg = segment(array_in)
>>> save(array_seg, "mysegment.nc")
```

If you ever find yourself in a situation where you seem to be hung and cannot get a new prompt, the interpreter is likely to be in a state where it is waiting for you to enter something specific (in its parse state). Hitting Ctrl-z will send a keyboard interrupt and should get you back to a prompt.

5.5.1 Syntax Highlighting

Your code is coloured as you type it in based upon the Python syntax. The colour scheme may depend upon your setup, but typically comments are displayed in green, strings in purple and Python keywords in blue.

5.5.2 Indentation

The Python scripting language uses indentation levels to define program blocks (rather than explicit `begin...end` syntax). The command line facilities provide limited automated support for indentation: here is an example that creates a function to produce factorials.

```
>>> def factorial(num):
        result = 1
        while num > 0:
                result = result + num
                num = num - 1
        return result

>>>
```

If you type in the above example you will find that when you hit Enter after the def clause, the next line is automatically indented for you (as you enter a new block). When within a block, you will be automatically placed at the same level as the previous statement (above, this happens for the num = num - 1 statement). Each press of the Backspace key will back you out of a level of indentation (above, it needs to be used before the return statement). After a pass, return, break, continue or raise statement you will automatically be backed out of a level of indentation. Hitting Enter when on the new line after `return result` will take you out of the function definition to a command prompt.

The `IDLE` IDE, provided for scripts (`file/new script` and `file/Open script` menus) provides fully automated support for indentation.

5.5.3 Command History

A feature that is worth knowing (because of the amount of typing it can save you) is the command history mechanism. Placing the cursor at the end of any line and pressing *Enter* will bring a copy of all the line. Alternatively, you can toggle through all previous commands sent to the interpreter by using Ctrl-↑ (or Alt-p) and Ctrl-↓ (or Alt-n). You can edit the line as desired before hitting the Enter key to have it sent to the interpreter.

5.5.4 Function Call Prompting

As soon as you type in the opening bracket for a function or method call a small box will pop up below the current line giving you a "tip" with regards to the arguments that are expected. For example, if the shell window contains:

```
>>> segment(
```

you will get a small highlighted window containing a tip on completing the statement:

```
segment(image, [quality=, shape=, stopping=, cooling=, looks=,
        maxiters=, regionsize=, intensity=])
```

together with summary information on the purpose of the routine and its default parameter settings. The window stays up until you enter the closing bracket. These tips appear for built-in functions, any functions or methods (including class constructors) from Python's library modules as well as for any function or methods defined by yourself. They do not, however, occur for list, tuple or dictionary methods.

For the functions or methods that you define, the tip will give the names of the parameters you define. The first (summary) line of the function or methods document string will also be displayed. This can be used to give information on the types of the arguments and return value.

5.5.5 Class Member Prompting

Once an object has been defined, typing the dot (".") command after the object name will bring up a list of possible members or methods available. For example, if the shell window contains:

```
>>> image, gis = load('myfile.nc')
>>> gis.
```

you will get a small scrolled window containing a list of the `gis` class members.

5.6 Developing *InfoPACK* Scripts

The interactive mode of calling *InfoPACK* modules is convenient when exploring a new dataset. For more advanced use, *InfoPACK* also provides the ability to define chains of operations (*scripts*).

InfoPACK allows two levels of scripts. At the simplest level, *InfoPACK* commands are stored in a text file which can be executed by the `Run Study` command from the GUI. For instance

```
segment.py image1 seg intensity=1 shape=3e-3

echo Display segmented image
openev seg

echo Test segmentation residual ratio
divide.py image1 seg ratio
stats.py ratio whole=1 intensity=1
```

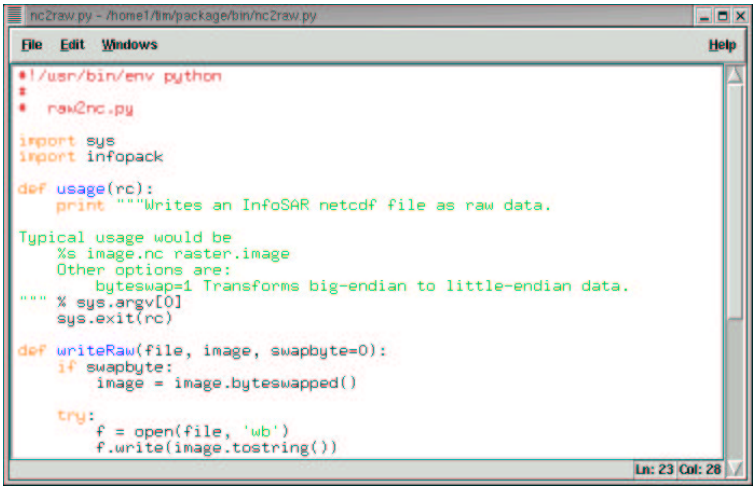

Figure 5.2. The IDLE Editor Window.

For those who need more control over their application, they can access the *InfoPACK* commands directly from Python. The names of these scripts often end with a `.py`. These can be prepared using any text editor, but *InfoPACK* also provides access to `IDLE`. This Open Source IDE provides a syntax-sensitive text editor and Python shell which is ideal for rapid development of Python scripts. The editor is described in section 5.7.

Python scripts are typically chains of *InfoPACK* commands, but may contain *any* Python or Numeric Python commands. Full documentation is accessible from the `Help` button.

5.7 The IDLE Integrated Development Environment

`IDLE` consists of the following integrated modules:

- *Python Shell Window*: gives you access to the Python interactive mode.
- *File Editor*: lets you create new or browse through and edit existing Python source files.
- *Path Browser*: for searching through the path of available module source files.
- *Class Browser*: for finding the methods of classes.
- *Debugger*: for the symbolic debugging of Python programs.

Documentation for `IDLE` is available at `http://python.org/idle/doc/`

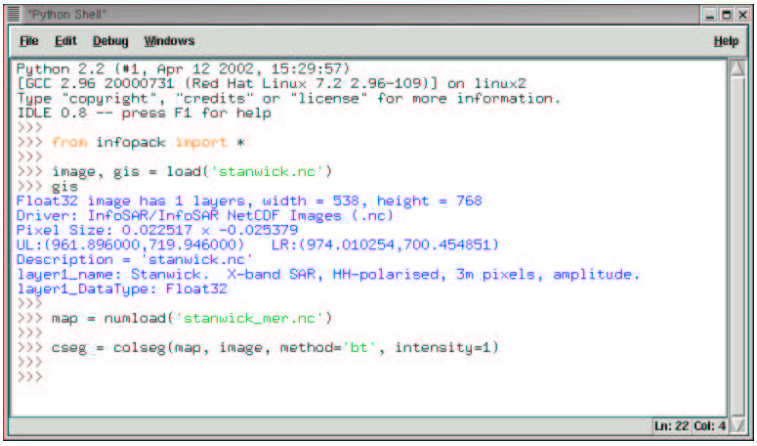
"Python Shell"
File Edit Debug Windows Help
Python 2.2 (#1, Apr 12 2002, 15:29:57)
[GCC 2.96 20000731 (Red Hat Linux 7.2 2.96-109)] on linux2
Type "copyright", "credits" or "license" for more information.
IDLE 0.8 -- press F1 for help
>>>
>>> from infopack import *
>>>
>>> image, gis = load('stanwick.nc')
>>> gis
Float32 image has 1 layers, width = 538, height = 768
Driver: InfoSAR/InfoSAR NetCDF Images (.nc)
Pixel Size: 0.022517 x -0.025379
UL:(961.896000,719.946000) LR:(974.010254,700.454851)
Description = 'stanwick.nc'
layer1_name: Stanwick. X-band SAR, HH-polarised, 3m pixels, amplitude.
layer1_DataType: Float32
>>>
>>> map = numload('stanwick_mer.nc')
>>>
>>> cseg = colseg(map, image, method='bt', intensity=1)
>>>
>>>
Ln: 22 Col: 4

Figure 5.3. The IDLE Shell Window with open script file.

Chapter 6

Case Studies

In this chapter we illustrate the capabilities of *InfoPACK* via some simple Case Studies.

6.1 Segmentation and Classification of Intensity Images

6.1.1 Introduction

In many remote sensing applications it is important to be able to classify SAR images in terms of their intensity. The difficulty is the effect of speckle noise which, as we shall see, makes assigning the intensity within a single pixel to a given class very inaccurate. The situation can be somewhat improved, in general, by averaging over a window of (say) 3x3 pixels. This averages out the edges between regions, which would be undesirable in a complicated scene or if small objects are of interest.

Provided that we can be certain that we have known (*ground truth*) boundaries that actually apply to the data, with complete homogeneity within each region, classification can be carried out by averaging the intensity within each ground truth region. This represents what we shall term the *Upper Limit* of classification capability. However, this method can only classify images with known regions; it has no ability to deal with raw imagery. Furthermore, in many situations the contents of the regions in an image,whether it's region boundaries are known or not, are not uniform. Crops, for example, grow at different rates in different parts of a field, and/or may not actually fill the specified region.

Segmentation provides a tool for deriving homogeneous segments within the data, based on the information within the data. The intensity within each segment then represents the most reliable information about the cross-section within that region. Inhomogeneities arising from any cause will be detected. In this study we compare the results of several possible approaches to the classification of regions, including segmentation. To ensure that we have ground truth to compare the results obtained, we use a simulated three–look SAR image containing a known pattern of fields. This is representative of the SAR imagery obtained with the ERS1 and ERS2 satellites.

Ground truth is available for this simulated image and is divided into a training and test set, illustrated in Figures 6.1.1(a) and 6.1.1(b).

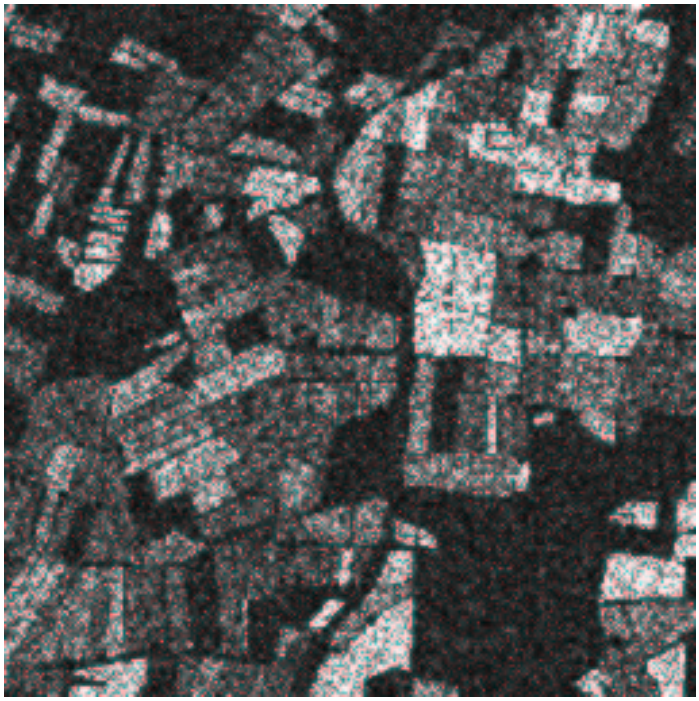

Figure 6.1. Simulated 3-look SAR data.

6.1.2 Comparison of classification methods

We can now compare the results obtained classifying the data in the test image using the following methods:

Upper Limit - classifying the average intensity in each ground truth region

Segmentation - classifying the average intensity in each segment

3x3 window - classifying the average intensity over a 3x3 window

Lower Limit - classifying the intensity in each pixel.

We expect the Upper Limit results (unachievable on "real" data) to be best; the Lower Limit results to be the worst. The major interest is in seeing how much better are the segmentation-based results than those achieved with a traditional local (3 by 3) filter.

In each case we calculate the likelihood that each region is a member of a gamma-distributed PDF of defined mean, depending on the class. The region is then assigned to that class which has the maximum likelihood. We assume for this comparison that the order parameter, determined by the speckle, is uniform across the image, as would be expected in practice.

The classification results are shown in Figure 6.1.2. Classification of a single pixel, shown in Figure 6.1.2(a) is clearly very poor, as expected. Indeed, the average probability of correctly classifying each category is only 0.364 and the associated confusion matrix, shown in Table 6.1.2(a) reveals how poorly the intermediate classes are assigned. Classification over a 3x3 window, shown in Figure 6.1.2(b) shows considerable improvement. However, the average fraction of correctly

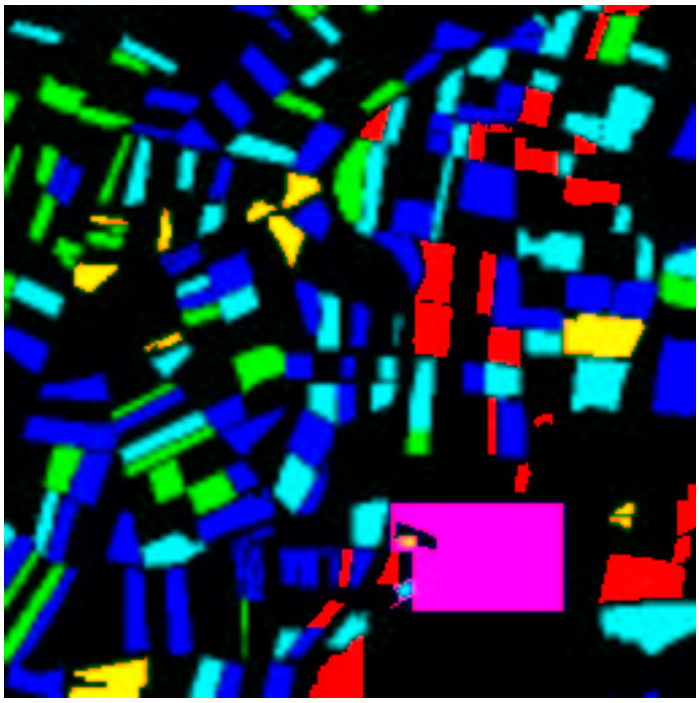

Figure 6.1.1(a): Training Ground Truth

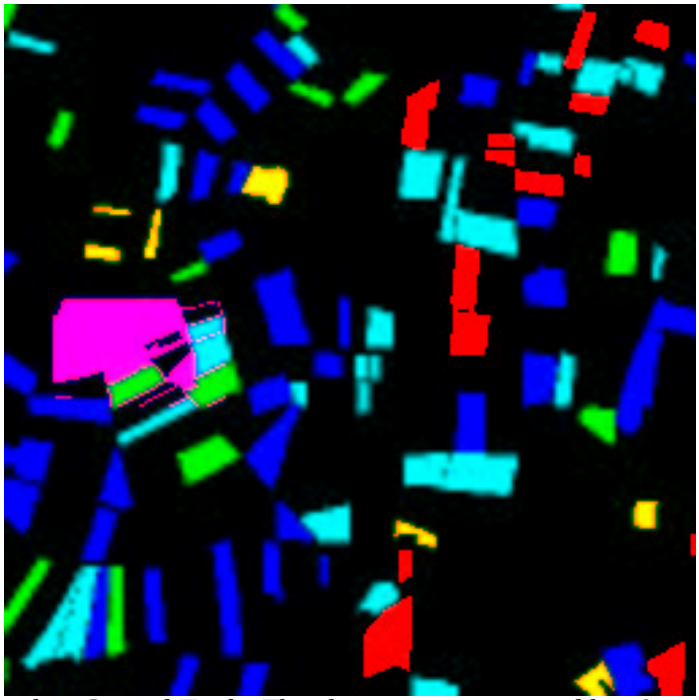

Figure 6.1.1(b): Test data Ground Truth. The classes are represented by: 0 - magenta; 1- blue; 2 - cyan; 3 - green; 4 - yellow; 5 - red.

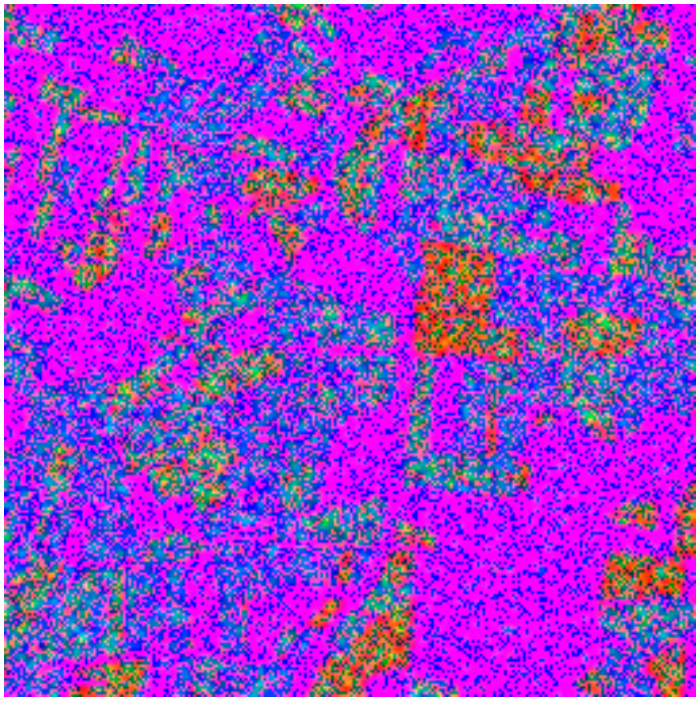

Figure 6.1.2(a): Classification using single pixels

classified pixels in each class is still only 0.571 and the associated confusion matrix in Table 6.1.2(b) reveals considerable spread in the intermediate classes.

The result of classification following segmentation, shown in Figure 6.1.2(c) is much closer to the ground truth, with the average fraction of correctly classified categories rising to 0.891 and the confusion matrix showing much less ambiguity between classes.

Finally, the Upper Limit, provided by the ground truth itself, leads to the results shown in Figure 6.1.2(d). The average probability of correct classification is now 0.993, and the confusion matrix shows very little ambiguity.

The conclusions are clear: if you have an accurate region map (ground truth) *use it*. If not: segment the image and classify the computed segmentation.

Table 6.1.2(a) Confusion Matrix for Single Pixel Classification

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.8000	0.3445	0.1699	0.0732	0.0362	0.0324
1	0.1777	0.3522	0.2712	0.1809	0.1447	0.0956
2	0.0203	0.1898	0.2280	0.2234	0.1338	0.1211
3	0.0020	0.0782	0.1525	0.1659	0.1646	0.1436
4	0.0	0.0248	0.0866	0.1300	0.1555	0.1258
5	0.0	0.0105	0.0919	0.2266	0.3653	0.4815

Table 6.1.2(b) Confusion Matrix for Classification using a 3 by 3 Filter

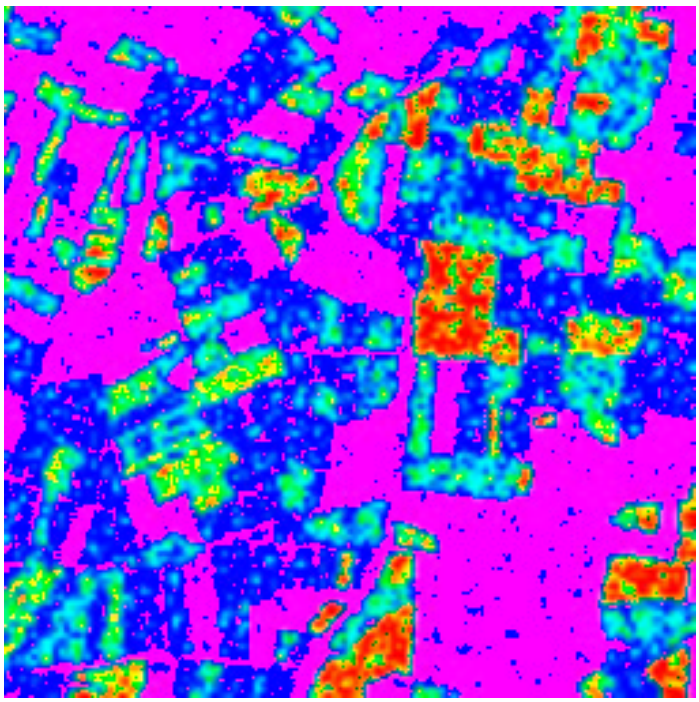

Figure 6.1.2(b): Classification using a 3 by 3 filter

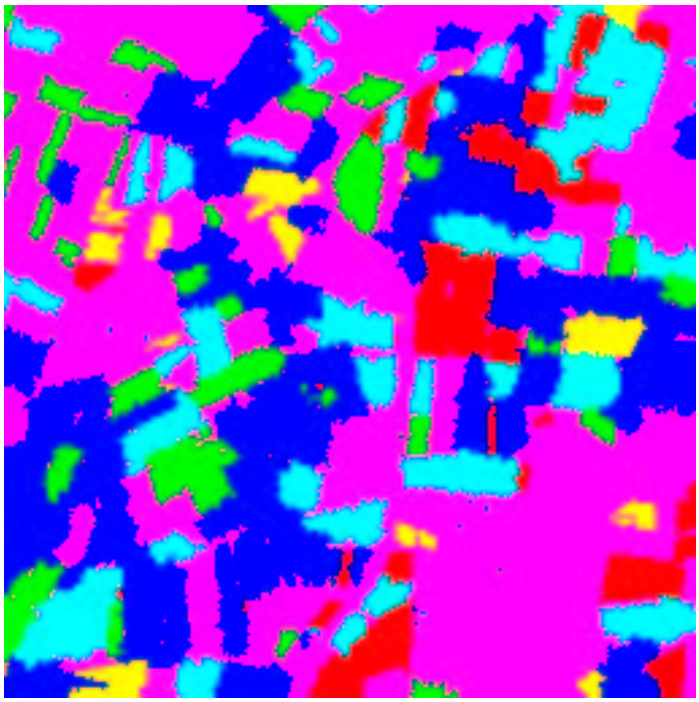

Figure 6.1.2(c): Classification using segmentation

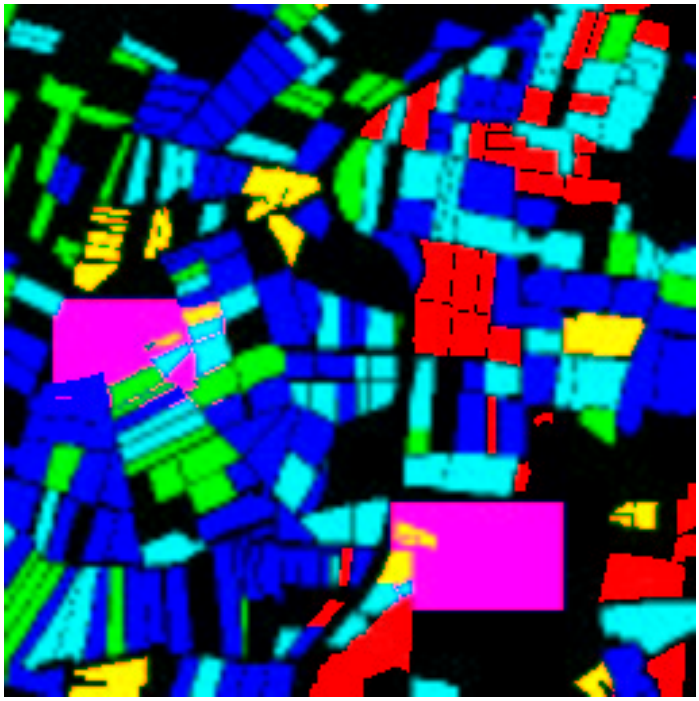

Figure 6.1.2(d): Classification from the Ground Truth

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.8196	0.0870	0.0059	0.0	0.0	0.0
1	0.1236	0.7941	0.2886	0.0614	0.0457	0.0032
2	0.0466	0.1177	0.5413	0.32440	0.2057	0.0442
3	0.0095	0.0012	0.1514	0.4286	0.2815	0.1627
4	0.0007	0.0	0.0128	0.1629	0.2998	0.2491
5	0.0	0.0	0.0	0.0227	0.2011	0.5408

Table 6.1.2(c) Confusion Matrix for Classification using Segmentation

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.9243	0.0472	0.0329	0.0222	0.0307	0.0099
1	0.0277	0.8908	0.1205	0.0258	0.0579	0.0099
2	0.0345	0.0501	0.7894	0.0490	0.0	0.0385
3	0.0122	0.0010	0.0495	0.8928	0.0	0.0005
4	0.0014	0.0003	0.0033	0.0039	0.9096	0.0
5	0.0	0.0005	0.0042	0.0052	0.0018	0.9405

Table 6.1.2(d) Confusion Matrix for Classification using Ground Truth

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.9899	0.0002	0.0	0.0	0.0	0.0
1	0.0095	0.9998	0.0	0.0	0.0	0.0
2	0.0007	0.0	1.0000	0.0	0.0	0
3	0.0	0.0	0.0	1.0000	0.0	0.0
4	0.0	0.0	0.0	0.0	0.9982	0.0287
5	0.0	0.0	0.0	0.0	0.0018	0.9713

6.1.3 Conclusions

It is apparent from this comparison that it is pointless attempting classification on individual intensity pixels in SAR. This example was for a 3 look system; with single look data the result would be much worse.

Averaging over a 3x3 window improves the result. The accuracy obtained in this way could be further improved by increasing the size of the window. However, there is then a problem that small regions will get averaged out, and disappear. What is required is a method which adapts the region size to the data; this is segmentation.

Segmentation followed by classification yields a considerable improvement, since it provides a filter whose window size and shape is adapted to the data. The fact that the segments are not matched exactly to the ground truth regions is a consequence of speckle noise. In this instance we know that the simulation was of a scene with uniform RCS within each region. The characteristic SAR speckle spoils this, and is the major hurdle to be overcome in handling SAR images.

The Upper Limit, as one would expect, provides the highest statistical accuracy. There are two important caveats to note, however:

1. The result is based on the assumption that the RCS within the region is indeed homogeneous. In real applications this will not generally be the case.

2. The ground truth can only be used where it exists, ie the true region boundaries are known. Segmentation, on the other hand, can use the training set to enable it to classify over as large an image as is required.

Thus, the overall conclusion is that segmentation provides an essential precursor to intensity classification. This is the approach taken by the classification routines in InfoPACK.

6.2 Segmentation and Classification of Multiple Intensity Images

6.2.1 Introduction

In the previous case study we demonstrated the capability of segmentation to improve the classification of a simulated 3-look SAR image. In some applications a sequence of such images is available. We assume as prior knowledge that every image in the sequence has fundamentally the same regions and boundaries with only variation in the RCS within each region. Under these conditions we may perform joint segmentation over every image in the sequence. If we do this, the common boundaries reinforce one another and help to overcome speckle noise. One would therefore expect this to be reflected in improved segmentation and classification performance.

To demonstrate this we generate another 9 simulated images, with the same underlying segments as those introduced in the previous study. These then differ only in terms of speckle; they simulate a set of images obtained by repeated passes of a SAR over the same scene. A similar analysis to that performed previously can be carried out on this set of intensity images. The ground truth training and test masks are identical to the previous analysis.

6.2.2 Comparison of classification methods

We now compare classification of the test image using the same methods as before:

- Upper Limit – classifying the average intensity in each ground truth region
- Joint Segmentation – classifying the average intensity in each segment
- 3x3 window – classifying the average intensity over a 3x3 window
- Lower Limit – classifying the intensity in each pixel.

Note that the averages are now taken over each pixel within each segment and over all images in the sequence. However, there is another option available for this particular case where the boundaries and underlying cross-section is identical in each image. The Maximum Likelihood reconstruction is obtained by averaging all the intensities, i.e. by summing the images in the first place. Only a single image is then needed for segmentation, rather than joint segmentation over the complete set. We shall call this method the High Limit, since it would be expected to be lower than the Upper Limit but (since it is a maximum likelihood algorithm) higher than joint segmentation [1]. In each case we calculate the likelihood that each region is a member of a gamma-distributed PDF of defined mean, depending on the class. The region is then assigned to that class which has the maximum likelihood. We assume for this comparison that the order parameter, determined by the speckle, is uniform across the image, as before.

The classification results are shown in Figure 6.2.2. Classification using a single pixel, shown in Figure 6.2.2(a) is still very poor. Indeed, the average probability of correctly classifying each category is only 0.673 and the associated confusion matrix, shown in Table 6.2.2(a) reveals major ambiguities in assigning the intermediate classes. Classification over a 3x3 window, shown in Figure 6.2.2(b) shows less improvement than might be expected, with the average fraction of correctly classified pixels in each class only 0.707. The associated confusion matrix in Table 6.2.2(b) again reveals considerable spread in the intermediate classes. Closer study of Figure 6.2.2(b) suggests that

[1] This latter expectation is not realised: we see below that joint segmentation provides the same performance, which is why there is not a specific routine for this case in `InfoLIB`

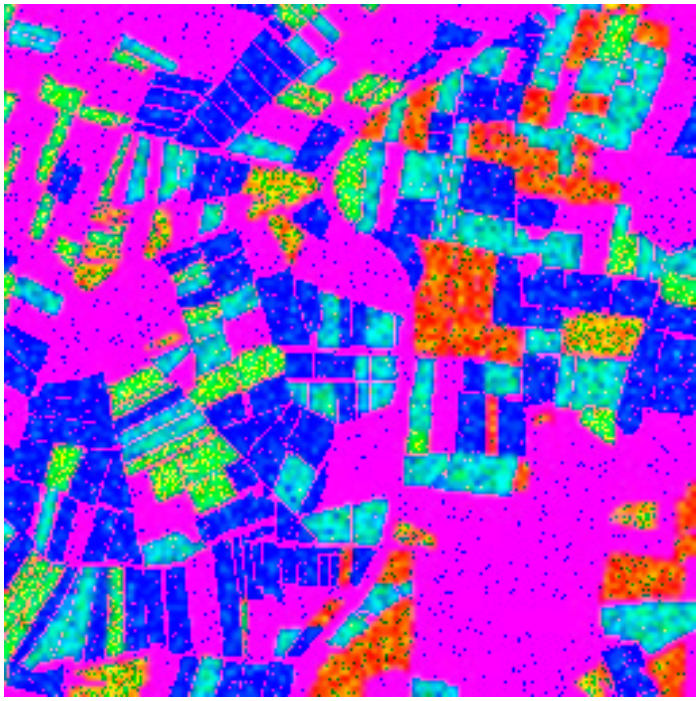

Figure 6.2.2(a): Classification using single pixels

the poor performance is due to the averaging filter. This means that each change in cross-section is surrounded by a band of intermediate value which are classified into other categories. The result of classification following segmentation, shown in Figure 6.2.2(c) is very close to the ground truth, with the average fraction of correctly classified categories rising to 0.984 and the confusion matrix showing much less ambiguity between classes. The Higher Limit result, shown in Figure 6.2.2(d), yields an identical value for the average probability of correct classification. The fact that there is no advantage over joint segmentation reveals that the latter is capable of extracting all the information within the set of identical images. Finally, the Upper Limit, provided by the ground truth itself, leads to the results shown in Figure 6.2.2(e). The average probability of correct classification is now 0.992 with the confusion matrix showing very little ambiguity.

Table 6.2.2(a) Confusion Matrix for Classification using Single Pixels

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.9696	0.0304	0.0	0.0	0.0	0.0
1	0.0346	0.8409	0.1236	0.0	0.0	0.0
2	0.0073	0.1455	0.6641	0.1821	0.0063	0.0
3	0.0	0.0085	0.2299	0.5036	0.2260	0.0320
4	0.0	0.0	0.0325	0.2767	0.4069	0.2839
5	0.0	0.0	0.0037	0.0695	0.2726	0.6543

Table 6.2.2(b) Confusion Matrix for Classification using 3 by 3 filter

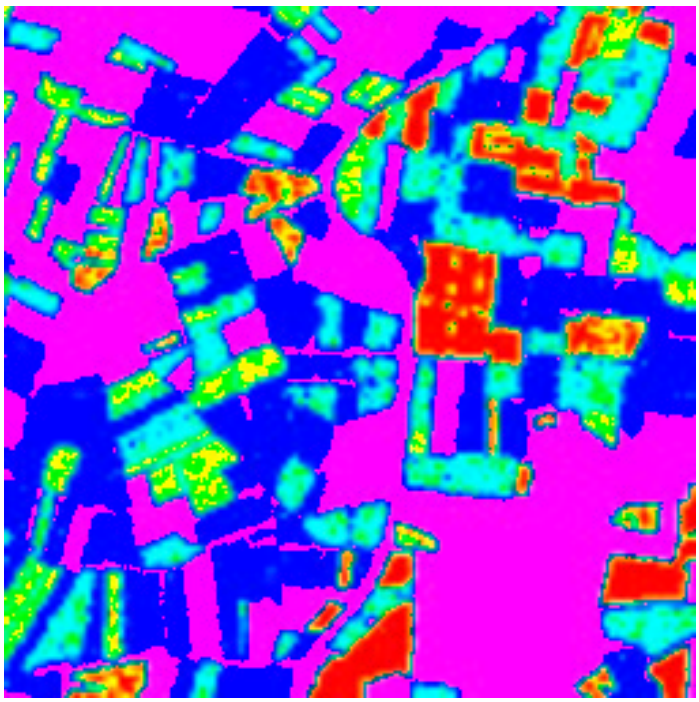

Figure 6.2.2(b): Classification using 3 by 3 filter

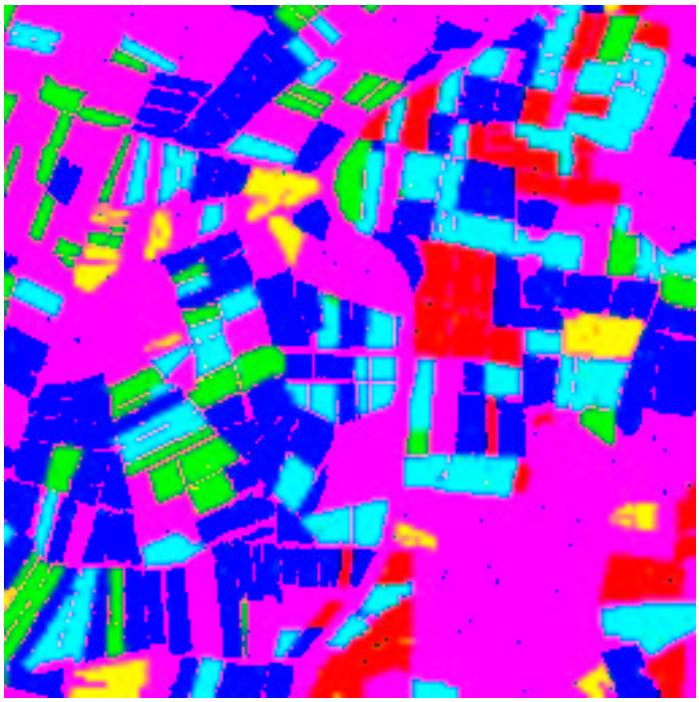

Figure 6.2.2(c): Classification using joint segmentation

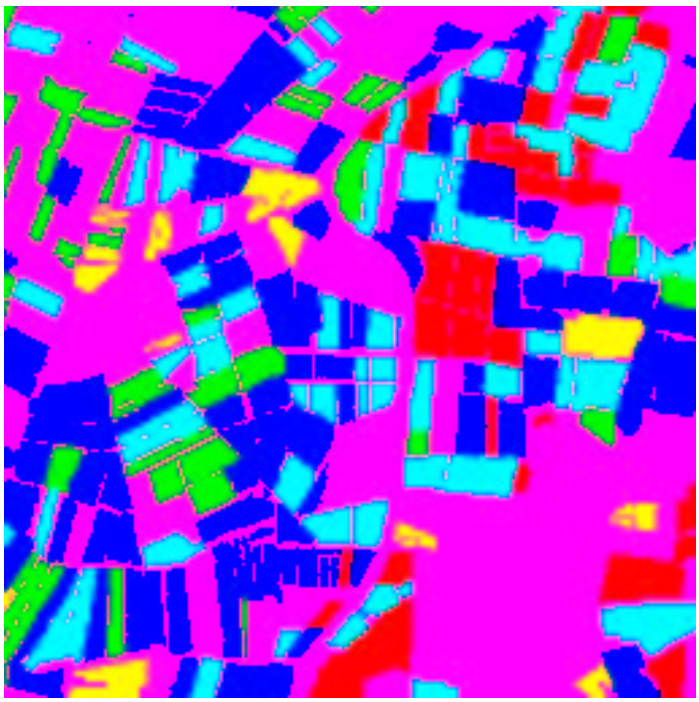

Figure 6.2.2(d): Classification using segmentation of the averaged images

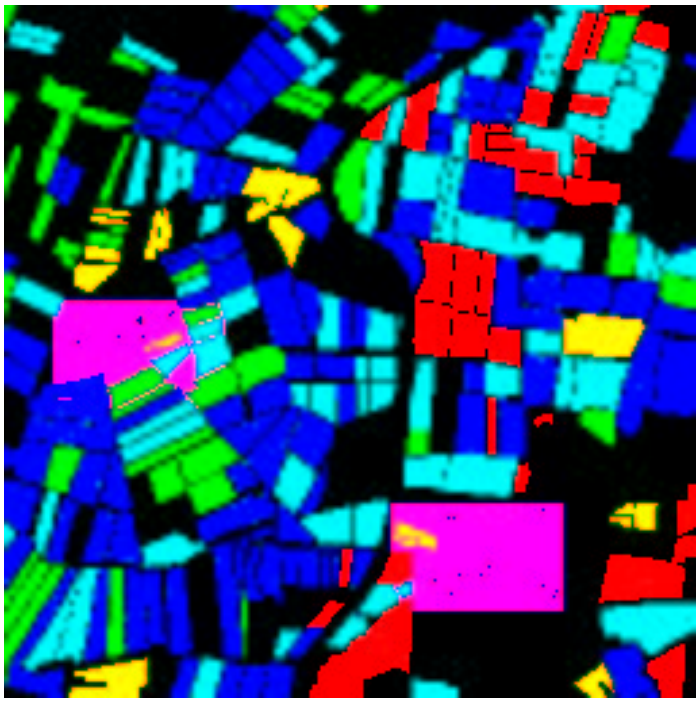

Figure 6.2.2(d): Classification using Ground Truth segments

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.8291	0.1020	0.0628	0.0070	0.0	0.0
1	0.0059	0.9769	0.0171	0.0	0.0	0.0
2	0.0	0.1004	0.7574	0.1394	0.0028	0.0
3	0.0	0.0060	0.1883	0.5374	0.2670	0.0013
4	0.0	0.0037	0.0640	0.2505	0.3766	0.3053
5	0.0	0.0	0.0047	0.0421	0.1864	0.7667

Table 6.2.2(c) Confusion Matrix for Classification using Joint Segmentation

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.9777	0.0176	0.0047	0.0	0.0	0.0
1	0.0064	0.9873	0.0062	0.0002	0.0	0.0
2	0.0006	0.0102	0.9823	0.0053	0.0014	0.0003
3	0.0	0.0020	0.0078	0.9830	0.0059	0.0013
4	0.0	0.0	0.0036	0.0054	0.9837	0.0072
5	0.0	0.0	0.0	0.0031	0.0084	0.9885

Table 6.2.2(d) Confusion Matrix for Classification using Segmentation of the averaged images

Assigned class: True class/Prob	0	1	2	3	4	5
0	0.9574	0.0257	0.0108	0.0061	0.0	0.0
1	0.0041	0.9948	0.0010	0.0	0.0	0.0
2	0.0003	0.0061	0.9934	0.0	0.0	0.0003
3	0.0	0.0013	0.0372	0.9615	0.0	0.0013
4	0.0	0.0018	0.0	0.0	0.9982	0.0
5	0.0	0.0	0.0	0.0	0.0	1.0

Table 6.2.2(e) Confusion Matrix for Classification using the Ground Truth segments

Assigned class: True class/Prob	0	1	2	3	4	5
0	1.0	0.0	0.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0	0.0	0.0
2	0.0	0.0003	0.9997	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.9992	0.0008
5	0.0	0.0	0.0	0.0	0.0	1.0

6.2.3 Conclusions

On comparing the results for classification over 10 images with the equivalent results for a single image we note that the classification performance is improved considerably for all except the 3x3 window result.

The 3x3 window method is shown to be a poor approach because the window introduces correlations at the boundaries of regions leading to misclassification. Joint segmentation followed by

classification yields a considerable improvement, since it provides a filter whose window size and shape is adapted to the data. The fact that the segments are not matched exactly to the ground truth regions is an inevitable consequence of speckle noise. In this instance we know that the simulated image had a uniform RCS within each region. The speckle statistics have of course spoilt this.

The High Limit uses the fact that the average intensity in every region is the same. This allows the average intensity to be segmented and classified. However, the performance is essentially the same as that achieved with joint segmentation, revealing that the latter is able to use all the information in the sequence of images. The Upper Limit, as one would expect, provides the highest statistical accuracy.

There are two important caveats to note, however.

1. This result is based on the assumption that the RCS within the region is indeed homogeneous. In real applications this will not generally be the case.

2. The "Upper Limit" classification can only be applied where region boundaries are known. It cannot extrapolate beyond the ground truth. Segmentation, on the other hand, can use the training set to enable it to classify over as large an image as is required.

Thus, the overall conclusion is as in Study 1: segmentation provides an essential precursor to intensity classification.

6.3 Change Detection between Intensity Images

6.3.1 Introduction

In the previous study we demonstrated the capability of segmentation to improve the classification of a set of simulated 3-look SAR images. Segmentation provides an adaptive function by maximising the segment size and matching the shape of the largest possible homogeneous region, consistent with the data. We introduced the prior knowledge that every image in the sequence has fundamentally the same regions and boundaries with only variation in the RCS within each region. Under these conditions we may perform joint segmentation over every image in the sequence. The common boundaries reinforce one another and help to overcome speckle noise. Thus the properties of each segment represent the most accurate estimates that can be obtained.

In some applications a sequence of images is available but we believe that although the underlying scene structure (segments) have *not* changed, the RCS will/may have changed between individual images. An example is a set of fields with growing crops: we want to monitor the growth between images. Then, we are concerned with detecting those regions that have changed in intensity.

Joint segmentation is ideally matched to this process since it constructs a set of segments that are common to each image. It is therefore possible to study the temporal development of identical regions of these images.

Two methods of change detection are considered here, neither of which requires any prior knowledge. Thus these methods are both unsupervised methods.

The first considers the sequence of images and detects whether the intensity within each region is constant throughout the sequence or not. This is based on temporal texture and uses the normalised log texture for the sequence in each element.

The second performs a maximum likelihood search for the position of a step change in each element over the sequence. It compares the likelihood of a step at all points with there being no change. Thus this offers additional information compared with the first method in that it both detects the presence of a change and determines its position in the sequence.

To demonstrate these approaches we simulate a sequence of ten images with six different classes, shown in Figure 6.2, whose cross-section varies from sample to sample as shown in Figure 6.3.

From Figure 6.3 we see that there is a change of -5 in the RCS of the red class at image number 4, one of 1 in the blue class at image number 5 and one of -2 in the green class at image number 6. The purpose of this study is to detect these changes.

6.3.2 Comparison of classification methods

As indicated above there are two different Maximum Likelihood tests that can be performed to test for changes. The first is to test the hypothesis that there has been a change in each region or not, at any point in the sequence. We shall call this change detection. The second method addresses the more sophisticated question of where any change is detected. Thus it provides information both on the existence and position of such a change. We shall name this positional change detection.

As before we shall compare results after the same four types of processing, namely:

Upper Limit: change detection on the average intensity in each ground truth region over the sequence

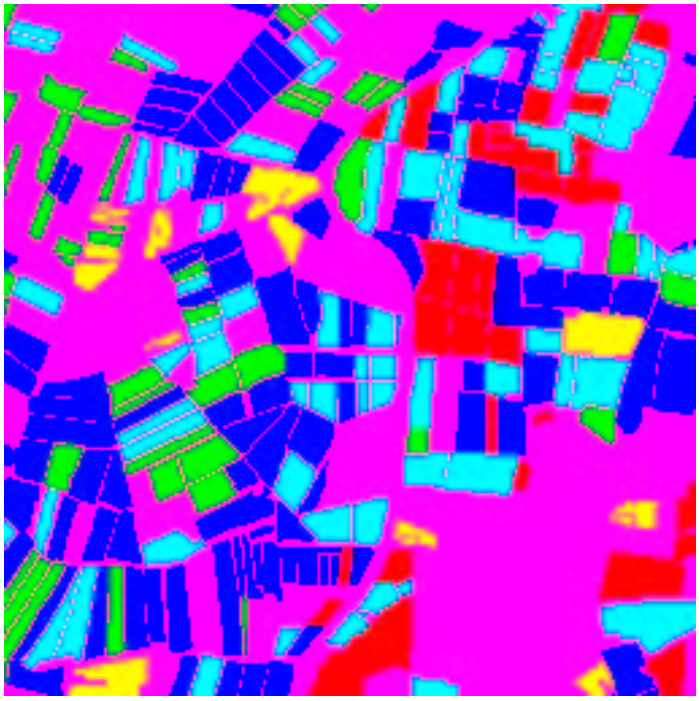

Figure 6.2. Test pattern of 6 defined classes used in this study for simulating image intensity.

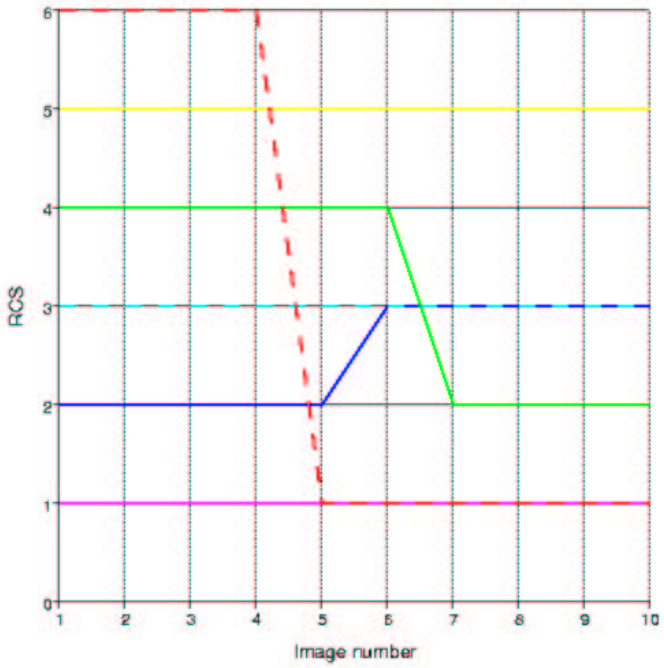

Figure 6.3. Plot of the mean intensity in each of the classes from Figure 6.2 during the sequence of images.

Figure 6.4. Predicted regions of change (white) and no change (black) in the image sequence.

Joint Segmentation: change detection on the average intensity in each segment over the sequence

3x3 window: change detection on the average intensity over a 3x3 window over the sequence

Lower Limit: change detection on the intensity in each pixel over the sequence.

Change detection

The output of this method is expected to be a map of those regions in the sequence where changes have been detected compared with those regions where no changes were found. Only two hypotheses are possible for the entire sequence. From figures 6.2 and 6.3 we note that changes are expected in the red, blue and green classes only. Thus the predicted changes are illustrated in Figure 6.4.

The ML test for homogeneity with a gamma-distributed PDF is given by the normalised log measure,

$$U = \overline{log(I)} - log(\bar{I}) \tag{6.1}$$

where the bars indicate averaging over all the pixels in a given segment for the complete sequence of images. If there is no change in RCS then the data is merely gamma distributed about its mean value with an order parameter equal to the effective number of looks (3 for these images). Any type of fluctuation in RCS will result in an increase in U. Thus changes can be detected by applying a suitable threshold to the normalised log image, formed from the sequence. The exact choice of threshold is selected to yield a reasonably low rate of false detections of change, in the absence of variations in the RCS.

When this analysis is applied to the sequence of images we obtain the results shown in Figure 6.8. If we compare the results with the expected ones we note that with single pixels (a) only

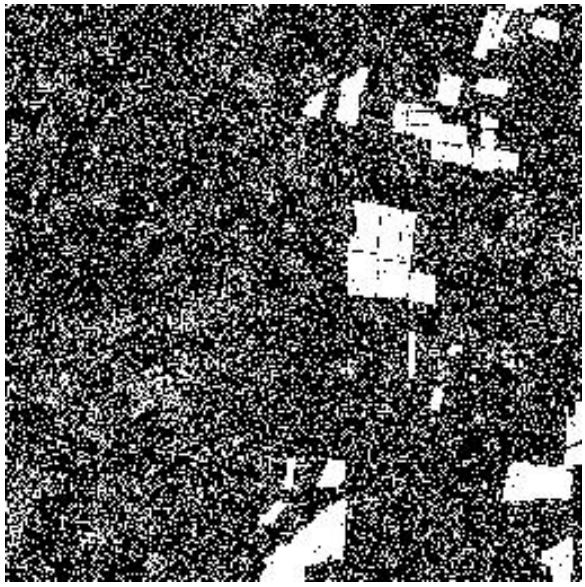

Figure 6.5. (a): Detected Changes using single pixels

the strongest change is visible, corresponding to the red class in Figure 6.2. After applying a 3x3 window (b), changes corresponding to the green class are visible. The result after joint segmentation (c) is considerably better with all classes being detected. Indeed, the changes resemble the predicted ones in Figure 3.

When the Upper Limit is considered, in which the predefined class boundaries determine the regions, the result is further improved so that the result is almost indistinguishable from Figure 6.4.

The performance comparison can be quantified in terms of the probability that changes are correctly detected, shown in Table 6.3.2.

Table 6.3.2: Probability of correct detection of changed regions for the different processing methods

Method	Pcor
Single Pixel	0.626
3 by 3 window	0.805
Joint Segmentation	0.971
Upper Limit	0.996

If we compare the probability of false classification (1-Pcor), we see that this progresses from 37% for single pixels to 19% for 3x3 window, 3% for joint segmentation and 0.4% for the Upper Limit. The considerable advantage offered by joint segmentation before change detection is obvious.

Positional change detection

Positional change detection offers the opportunity to detect both the presence of a change and its position. The likelihood that a change occurs at a position K in a sequence can be shown to be given by

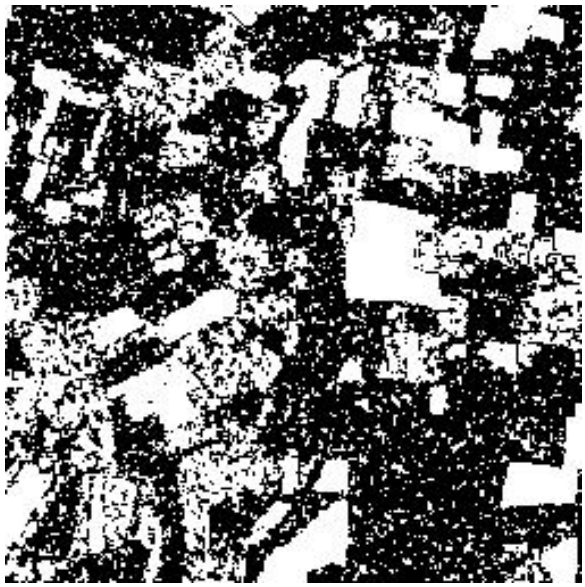

Figure 6.6. (b): Detected Changes using a 3 by 3 filter

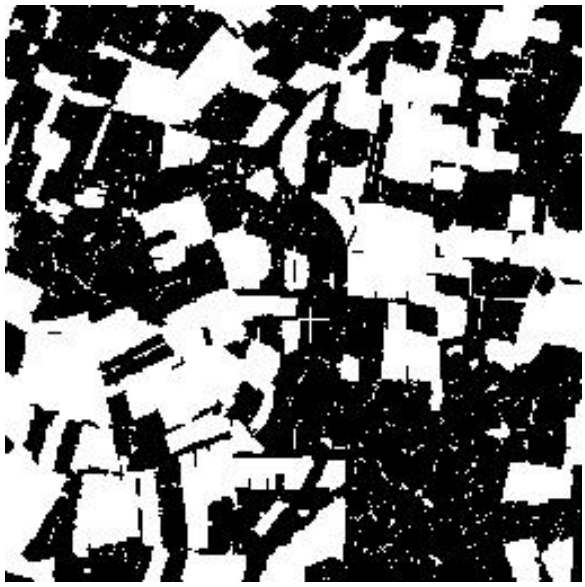

Figure 6.7. (c): Detected Changes using joint segmentation

Figure 6.8. (d): Detected Changes using Prior Knowledge

$$(K) = -Klog(\overline{I_1}) - (M-k)log(\overline{I_2}) + Mlog(\overline{I_0})$$

where the bars denote the average values in each segment over the first K images (region 1), the second $M-K$ images (region 2) and all M images (region 0) respectively. Once again we can apply a threshold to ensure that the detected change is unlikely to be caused by speckle noise when no change is expected.

The predicted pattern of changes is expected to be given by the red, green and blue classes in Figure 6.2. The actual detections are shown in Figure 6.3.2. The single pixel results (a) show that only the red change, with an RCS step of 5, is reasonably detected. After a 3x3 window (b) the green change, with step of 2, is reasonably well defined and the blue change, with a set of 1, is partially detected. Following segmentation (c), all the changes are well detected. Finally, the Upper Limit shows almost no difference from the expected changed classes in Figure 6.2

Once again, the most useful comparison is in terms of the detection probabilities, as summarised in Table 6.3.2. The probability of correctly detecting the change at the correct position in the sequence is listed for the four different processing methods. The average result over all 8 possible positions is compared with the specific results for the three positions (4, 5, and 6) where change is predicted.

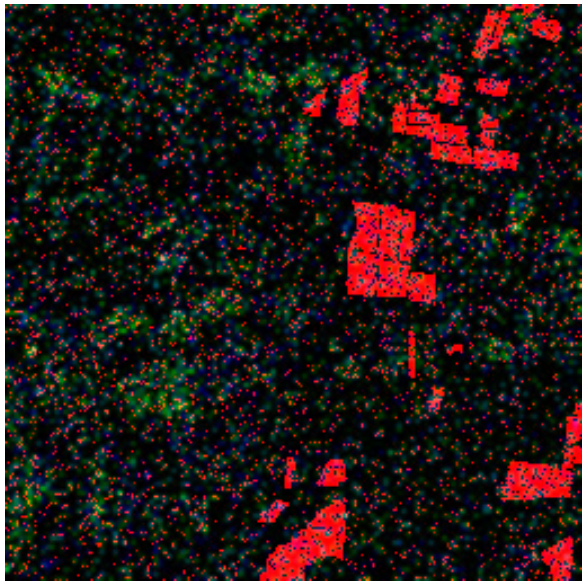

Figure 6.3.2(a): Detected Changes using Single Pixels

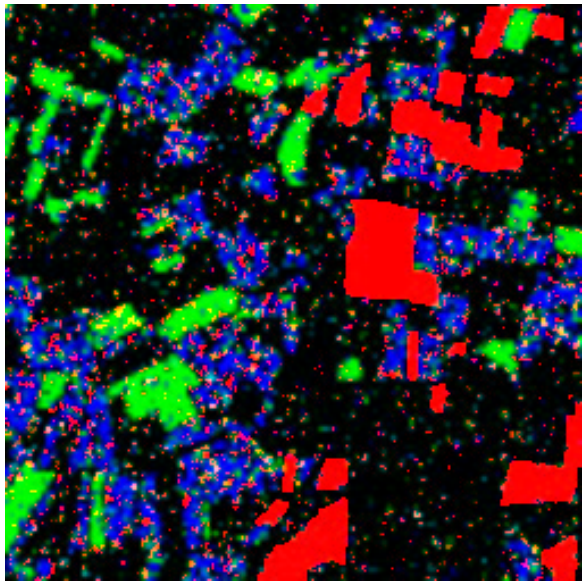

Figure 6.3.2(b): Detected Changes using 3 by 3 filter

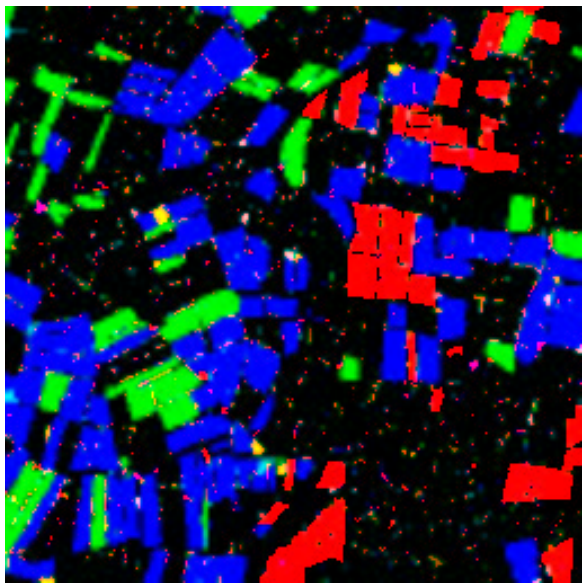

Figure 6.3.2(c): Detected Changes using Joint Segmentation

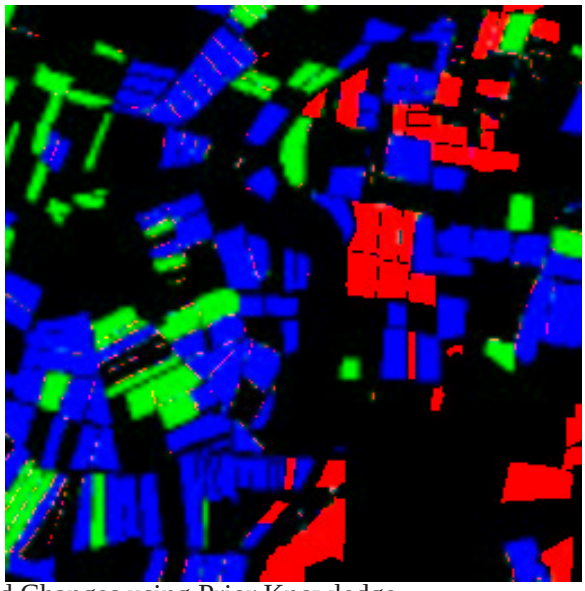

Figure 6.3.2(d): Detected Changes using Prior Knowledge
The colours indicate detected changes at the different positions within the image sequence. Brown (position 1), magenta (2), cyan (3), red (4), blue (5), green (6), yellow (7) and white (8).

Table 6.3.2: Probability of detecting changes at the correct position within a sequence. The average probability is calculated over all 8 possible positions. The specific results for the known change positions are also listed.

Method	Average	Position 4	Position 5	Position 6
Single Pixel	0.8618	0.9004	0.5123	0.5640
3 by 3 window	0.9433	0.9782	0.7322	0.8970
Segmentation	0.9925	0.9944	0.9729	0.9897
Upper Limit	0.9996	0.9998	0.9991	0.9995

The single pixel results confirm the fact that only the strong change (4) is reasonably detected. The average false classification probability is 14%. The 3x3 window gives better response for all three positions of change, with the results graded in order of step strength (4, 6 and 5) and an average false classification probability of 5.7%. Again, there is a considerable improvement using joint segmentation with an average false classification of only 0.75%. The individual detection probabilities are again in order of step strength. Finally, the Upper Limit is almost exact with an average false classification probability of only 0.04%. Once again the individual detection probabilities are in order of step strength.

In making these comparisons we observe the advantage of using defined edge maps for the regions of interest, where these are available and can be trusted. However, it must be noted that agricultural regions, such as fields, are very unlikely to be completely homogeneous. The Upper Limit offers no capability for detecting these inhomogeneities. Furthermore, they will tend to degrade so-called Upper Limit results, whereas they will have much less impact on joint segmentation. Perhaps most important is the fact that depending on existing edge maps excludes any possibility of extrapolating beyond available map data. With segmentation extrapolation is automatic.

Finally, it is instructive to compare the detection probabilities in the two methods of change detection. From this comparison one would deduce that the detection of change together with its position is more sensitive than the simple change detection process. Since positional change detection offers additional information as well as being more sensitive, this is the technique that should be preferred.

6.3.3 Conclusions

- Joint segmentation offers a valuable precursor to change detection because it adapts the region size to the data and presents a common set of boundaries for the whole sequence of images.
- The Upper Limit provides the most sensitive method for detecting changes. However, it suffers from three disadvantages:
  - It cannot detect inhomogeneities within the defined regions.
  - The sensitivity is degraded by any inhomogeneities.
  - It only process within defined regions, with no extrapolation capability such as that possessed by segmentation.
- Joint segmentation is somewhat less sensitive than the Upper Limit but is considerably better than using a 3x3 window, which in turn is better than single pixels.
- Positional change detection is more sensitive than change detection alone and offers additional important information; namely, the position in the sequence of any change detected.

Therefore we reccommend that change detection should always adopt the simultaneous detection of change and its position.

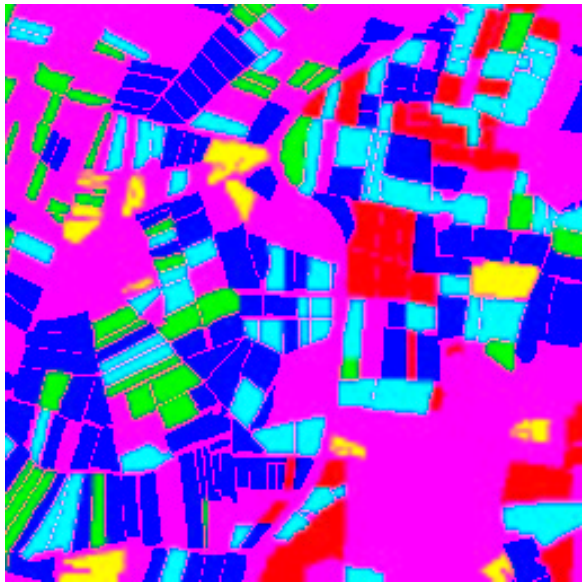

Figure 6.9. Test pattern of 6 defined classes used in this study for simulating image intensity.

6.4 Temporal Signatures of Intensity Images

6.4.1 Introduction

In the previous case study we demonstrated the capability of segmentation to improve the classification of multiple simulated 3-look SAR images. Here we extend this study to the matching of regions to a known temporal signature - a practical example would be the delineation of a set of different crops whose growth pattern is known. This growth pattern (signature) will often in practice be derived from a previous data or (as in this model study) from a defined training mask so that the process is one of supervised classification.

Joint segmentation is well matched to this process since it constructs a set of segments that are common to each image. It is therefore possible to study the temporal development of identical regions of these images.

In this study we simulate a sequence of ten images with the same six classes as in the previous study, shown in Figure 6.9, but with cross-section variation from sample to sample as shown in Figure 6.10.

From Figure 6.10 we see that, with the exception of the maroon class, all the classes show a variation in RCS over the sequence. Typical variations include a gradual increase, analogous to crop growth, for example, and a sudden drop, analogous to the reaping of crops. The purpose of this study is to classify regions in the sequence of images in terms of this signature. In fact, we do not use this signature directly, but train the temporal signature of each class using the training masks shown in Figure 6.11(a). The validation is then performed by testing over the regions denoted by the test masks in Figure 6.11(b).

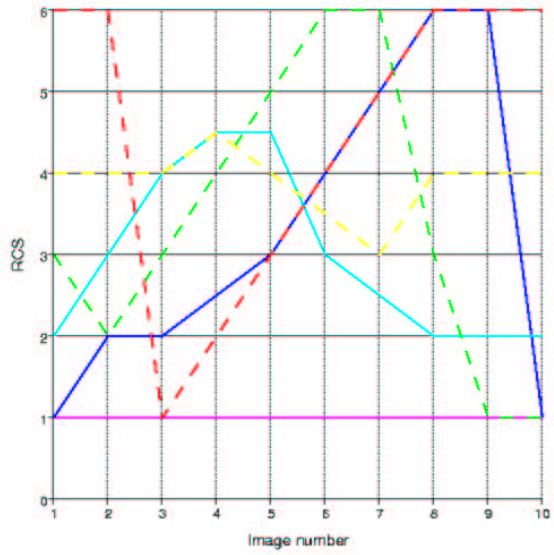

Figure 6.10. Plot of the mean intensity in each of the classes from Figure 6.9 during the sequence of images

6.4.2 Classification methods

As before we simulate a series of images having random 3-look speckle with the underlying RCS shown in Figure 6.10. We then perform Maximum Likelihood temporal signature classification on these images, after different processing techniques have been applied, namely:

Upper Limit: classification on the average intensity in each ground truth region over the sequence

Joint Segmentation: classification on the average intensity in each segment over the sequence

3x3 window: classification on the average intensity over a 3x3 window over the sequence

Lower Limit: classification on the intensity in each pixel over the sequence.

The first stage is to perform the pre-processing stage described above. Next we train the temporal signature by estimating the properties of each class over the regions defined by the training mask in Figure 6.11. these properties are next used to provide Maximum Likelihood classification over the different temporal signatures. The underlying model for the speckled data is that it is gamma-distributed with order 3, corresponding to 3-look SAR, and varying mean from image to image. The likelihood we wish to calculate therefore corresponds to data having order 3 and pre-trained mean over the sequence. The likelihood of each segment belonging to each of the temporal signature categories can be calculated. The ML solution for each segment then denotes the assigned class for that segment.

When this analysis is applied to the sequence of images we obtain the results shown in Figure 6.4.2. Corresponding average probabilities of correct classification are summarised in Table 6.4.2. With single pixels (a) the classification is clearly similar to the expected result in Figure 6.9. However, there are spots of misclassified data visible, caused by speckle noise. The average probability

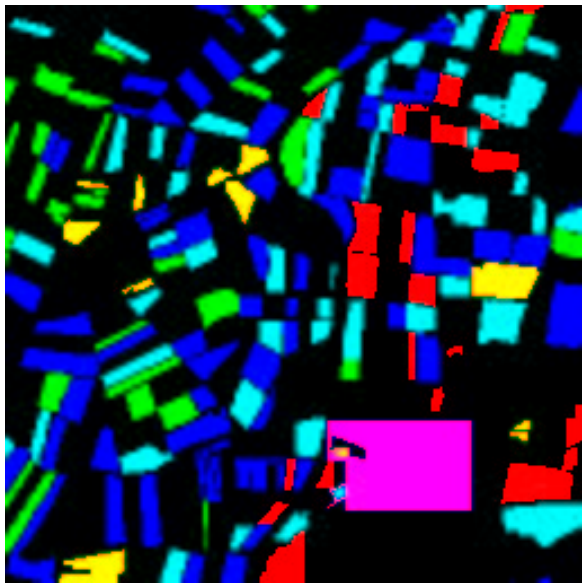

Figure 6.11. (a) Ground truth training masks for the simulated data. The classes are represented by: 0 - magenta; 1- blue; 2 - cyan; 3 - green; 4 - yellow; 5 - red.

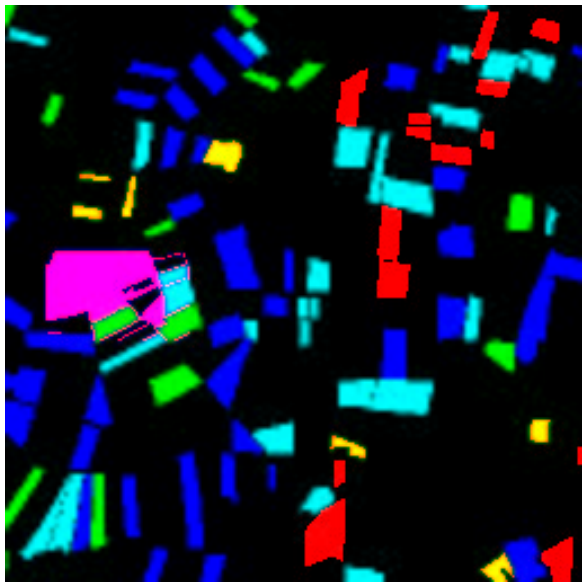

Figure 6.12. (b) Ground truth test masks for the simulated data.

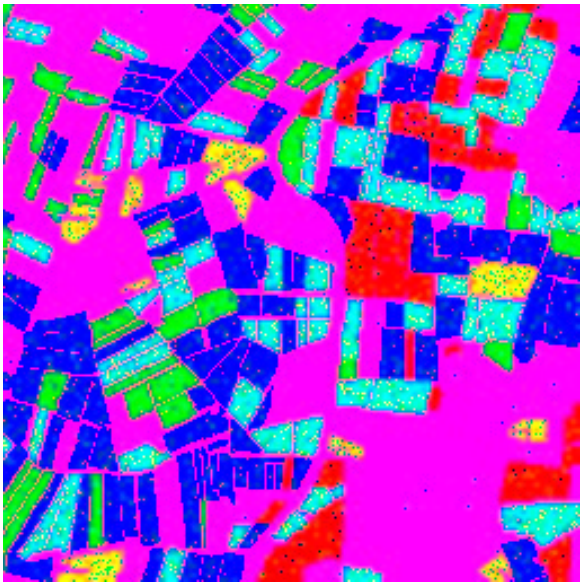

Figure 6.4.2(a): Classification using single pixels

of correct classification is 0.899. After averaging over a 3x3 window, the result in (b) shows that these spots have been averaged out. However, the correlations introduced by the filter leave a ring of misclassified regions around each large area. Thus, although the probability of correct classification is reasonable within large areas of homogeneous RCS, the average for this test scene, 0.954, is not as good as one might expect. Indeed, an image with more structure would yield an even worse result. The result after joint segmentation (c) is considerably better with all classes being separated, including most of the narrow sections between large areas. The average probability of correct classification has risen considerably, to 0.992. Finally, the Upper Limit yields essentially perfect classification.

Figure 4: Classified results from (a) single pixels, (b) 3x3 window, (c) joint segmentation, (d) Upper Limit.

Table 6.4.2: Average probability of correct classification for the different processing methods

Method	Pcor
Single pixel	0.899
3 by 3 Window	0.954
Joint Segmentation	0.992
Upper Limit	1.000

If we compare the probability of false classification (1-Pcor), we see that this progresses from 10% for single pixels to 4.6% for 3x3 window, 0.8% for joint segmentation and 0% for the Upper Limit. The considerable advantage offered by joint segmentation before classification is obvious.

As a final comparison we show the confusion matrices for the different methods in Table 2. Generally speaking the classification results are better than those for the 10 scenes considered when we introduced joint segmentation and classification. This is because the average RCS in each class is higher than before.

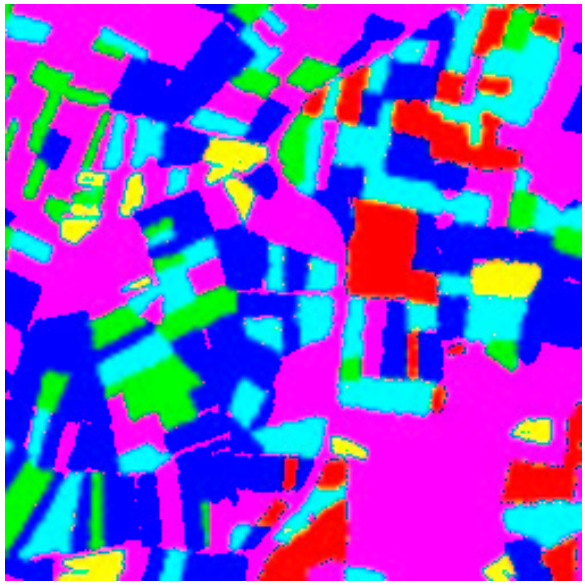

Figure 6.4.2(b): Classification using 3 by 3 filter

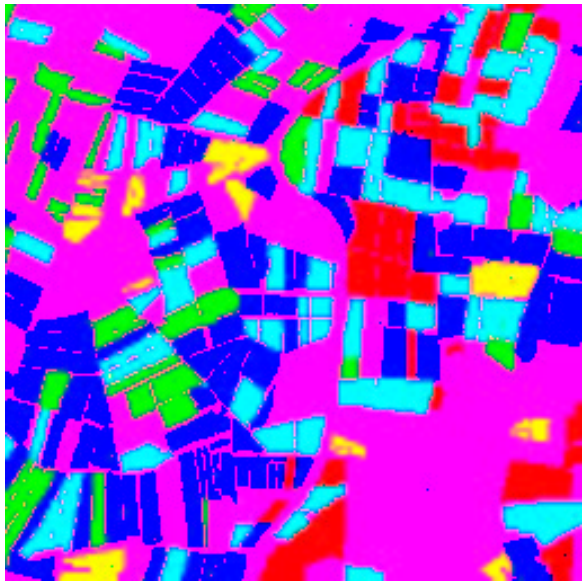

Figure 6.4.2(c): Classification using Joint Segmentation

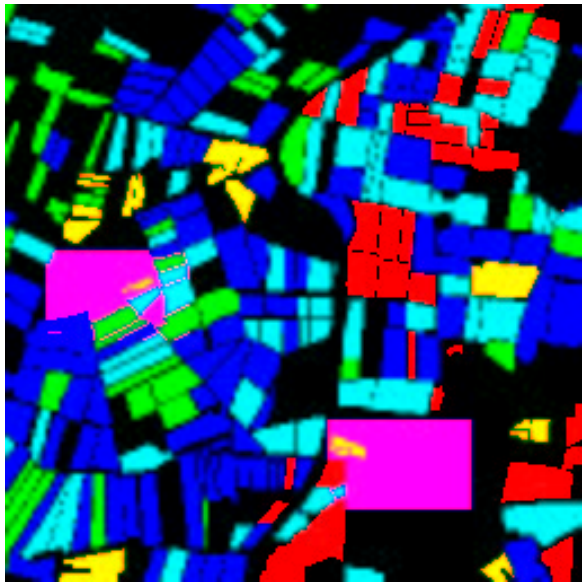

Figure 6.4.2(d): Classification using Upper Limit

These detailed results bear out the observations made from the classified images. In particular, the averaging window seems to have the most serious degrading effect on the classification of class 0. This is because all the other classes tend to be averaged down toward this class.

Table 6.4.2 Confusion matrices for temporal signature classification using the different processing methods. (a) single pixel, (b) 3 x3 window, (c) joint segmentation (d) Upper Limit

Table 6.4.2(a): Single Pixel

	Assigned class					
True Class	0	1	2	3	4	5
0	0.9993	0	0.0007	0	0	0
1	0.0009	0.9404	0.0215	0.0245	0.0072	0.0055
2	0.0017	0.0210	0.8290	0.0706	0.0761	0.0017
3	0	0.0170	0.0660	0.9059	0.0105	0.0007
4	0	0.0072	0.1193	0.0199	0.7811	0.0723
5	0	0.0104	0.0042	0.0026	0.0439	0.9389

Table 6.4.2(b): 3 by 3 Window

	Assigned class					
True Class	0	1	2	3	4	5
0	0.8581	0.0351	0.0689	0.0372	0.0007	0
1	0.0002	0.9984	0.0012	0	0.0002	0
2	0.0006	0.0014	0.9802	0	0.0178	0
3	0	0.0060	0.0060	0.9880	0	0
4	0	0	0.0823	0	0.9177	0
5	0	0	0.0005	0	0.0179	0.9816

Table 6.4.2(c): Joint Segmentation

	Assigned class					
True Class	0	1	2	3	4	5
0	0.9723	0.0041	0.0122	0.0115	0	0
1	0.0012	0.9960	0.0015	0.0007	0.0003	0.0002
2	0.0003	0.0019	0.9972	0.0003	0	0.0003
3	0	0.0013	0.0026	0.9961	0	0
4	0	0.0018	0.0018	0	0.9964	0
5	0.0005	0.0010	0.0010	0	0.0010	0.9963

Table 6.4.2(d): Upper Limit

	Assigned class					
True Class	0	1	2	3	4	5
0	1.0000	0	0	0	0	0
1	0	0.9998	0.0002	0	0	0
2	0	0.0003	0.9997	0	0	0
3	0	0	0	1.0000	0	0
4	0	0	0	0	1.0000	0
5	0	0	0	0	0	1.0000

6.4.3 Conclusions

- Joint segmentation offers a valuable precursor to temporal signature classification because it adapts the region size to the data and presents a common set of boundaries for the whole sequence of images.
- The Upper Limit provides the most sensitive method for classification. However, it suffers from three disadvantages:
  - It cannot detect inhomogeneities within the defined regions.
  - The sensitivity is degraded by any inhomogeneities.
  - It only processes within defined regions, with no extrapolation capability such as that possessed by segmentation.
- Joint segmentation is somewhat less sensitive than the Upper Limit but is considerably better than using a 3x3 window, which in turn is better than single pixels.
- The 3x3 window method suffers from edge effects due to averaging over the boundaries.
- Therefore temporal signature classification should adopt joint segmentation as an initial pre-process.

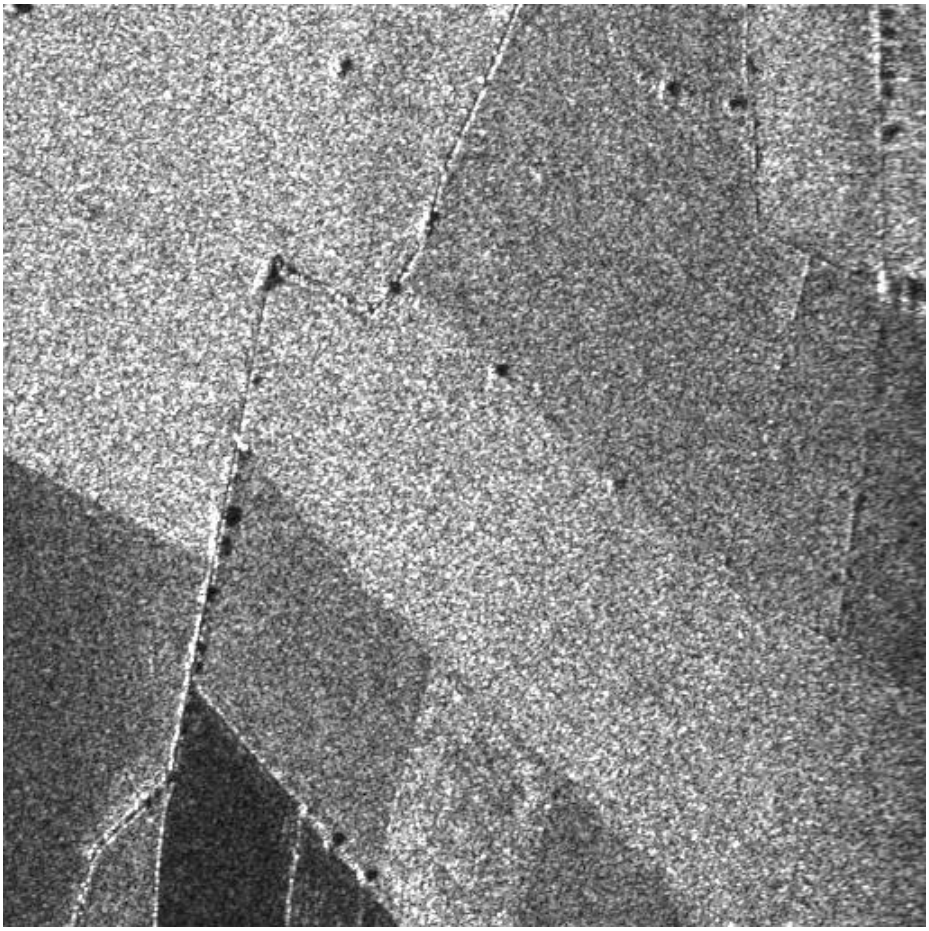

Figure 6.13. Typical HH SAR image obtained from the DLR ESAR system.

6.5 Handling Correlated Images

6.5.1 Introduction

On many occasions SAR images are sampled at spacing less than their resolution. Indeed, Nyquist sampling of the detected intensity image would require spacing of half the resolution in each direction to retain all the information in the image. Figure 6.13 illustrates part of a typical SAR image obtained using the DLR ESAR system. In this mode of operation the range resolution is 4 m, the azimuth resolution is 1.8 m, the pixel spacing is 1.25 m in range and 0.68 m in azimuth. The effective number of looks (ENL) is specified to be 4.0. Based on this information we note that the image is over sampled with 3.2 samples per resolution length in range and 2.6 in azimuth. The effect of this oversampling is to make the speckle, which reflects the coherent interference between random scatterers within each resolution cell, subtend approximately 3x3 pixels.

However, as noted previously the segmentation process implicitly assumes that neighbouring pixels are not correlated. Speckle correlation will therefore impact on the quality of the segmentation reconstruction. In this case study we demonstrate the effect of such correlations on segmentation and despeckling, and show how the problem can be overcome.

6.5.2 Correlation and Segmentation

Previous studies have determined suitable sets of parameters for intensity segmentation. In Figure 6.5.2(a) we show a detail from Figure 6.13, together with its segmentation, using default parameters, in Figure 6.5.2(b). It is immediately apparent that the segmented image has far too many segments, which appear to follow the speckle fluctuations. This is indeed precisely the effect that correlated speckle has on segmentation. A region of high speckle intensity, subtending several pixels, is reconstructed as a single segment. The process can be understood in terms of a simple multiplication noise speckle model.

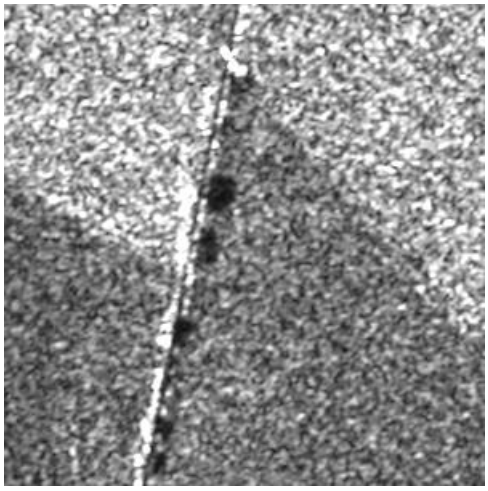

Figure 6.5.2(a)

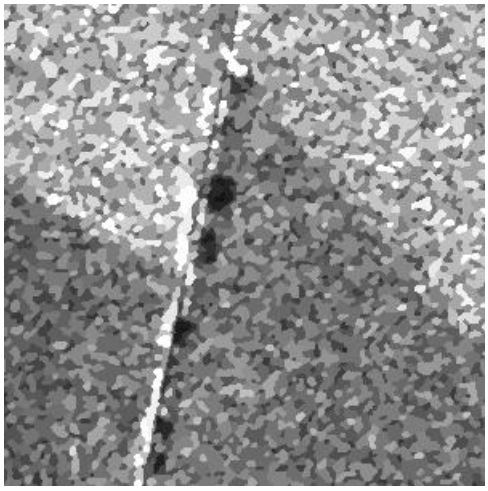

Figure 6.5.2(b)

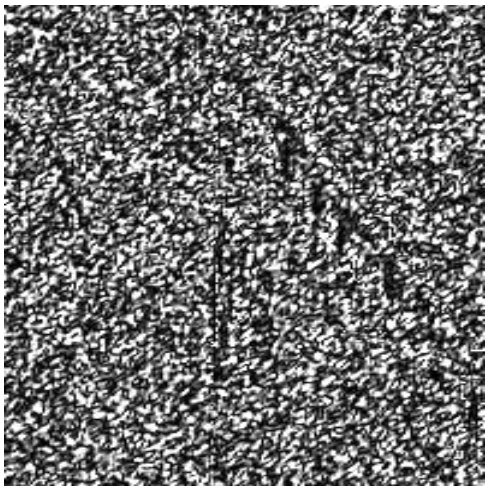

Figure 6.5.2(c)

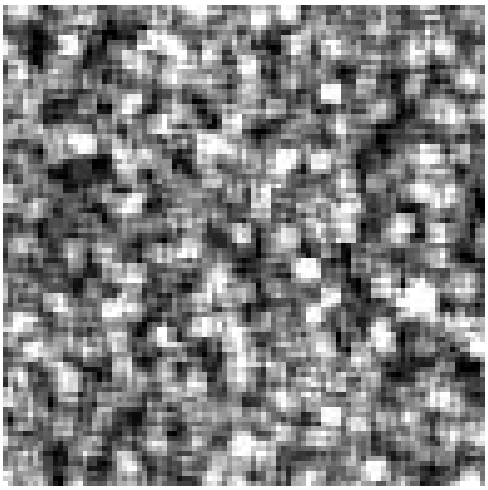

Figure 6.5.2(d)

6.5.2(a) A detail from the original image shown in Figure 6.13 (b) The segmentation (c) The residual speckle (d) The expected residual speckle

Suppose that the observed intensity is represented as a speckle noise model such that

$$I = \sigma \times n$$

where σ is the underlying cross-section and n the multiplicative noise, which is gamma-distributed with order L, where L is the ENL. Let us suppose that the segmentation succeeds in representing the

underlying cross-section exactly. Under these conditions the variance of the intensity normalised by the cross-section is given by

$$\frac{var(I/\sigma)}{< I/\sigma >^2} = \frac{var(n)}{< n >^2} = \frac{1}{L}$$

Thus, if we take the ratio of the original intensity to the derived segmentation, the result should have the properties of random speckle with the expected correlation properties and relative variance. In Figure 6.5.2(c) we show the residual speckle, derived from Figures 6.5.2(a) and 6.5.2(b). In Figure 6.5.2(d) we show a simulated speckle image having approximately the correct correlation properties. The correlation in 6.5.2(d) is assumed to correspond to a square averaging window; no attempt has been made to reproduce the correct correlation function. It is clear that 6.5.2(c) fails to show the expected speckle correlation properties, a consequence of the segmentation tending to reconstruct the individual speckle blobs. This is further borne out by the relative variance of Figure 6.5.2(c), which is 14.7, compared with the ENL of 4.0.

6.5.3 Speckle correlation properties

The SAR image in Figure 6.13 was of a detected (amplitude) image. In some cases the single-look-complex (SLC) image is available. In this case we can perform a detailed study of the instrument correlation properties, as distinct from any cross-section fluctuations. Consider the autocorrelation function (ACF) of the received complex field at lag k, which is given by

$$g_E(k) = \frac{< E_0 E_k^* >}{\sqrt{< E_0 E_0^* >< E_k E_k^* >}} = \frac{< E_0 E_k^* >}{< I >}$$

provided the field, E, is a stationary ergodic random process. Thus constructing the ACF of the complex field yields a correlation function with a value of 1.0 at zero lag and decaying to zero as the lag increases, provided the field has zero mean. In fact, $g_E(k)$ corresponds to the ACF of the instrument function so that

$$g_E(k) = h(0)h^*(k)$$

If the SLC image is detected and the ACF of the resulting intensity constructed, this will be given by

$$g_I(k)m = \frac{< I_0 I_k >}{< I_0 I_k >} = \frac{< I_0 I_k >}{< I^2 >} + 1$$

This can be split up into four separate terms. Firstly, there is the flat background of $< I^2 >$. Secondly, there is the interference term corresponding to random scatterers within the instrument width. This would be the response if the cross-section were uniform. In this case the intensity ACF would be give by

$$g_I(k) \approx 1 + \frac{1}{L} \mid g_E(k) \mid^2$$

This has the same shape as the square modulus of the complex field ACF defined above. Thirdly, there is a term corresponding to incoherent imaging of the cross-section fluctuations. The fourth term represents coherent interference between random scatterers modulated by the cross-section

fluctuations. This approximates to the square modulus of the complex field ACF, if the cross-section correlation function is approximately constant over the decay length of the instrument correlations. The total intensity ACF can then be approximated by

$$g_I(k) \approx 1 + \frac{1}{L} \mid g_E(k) \mid^2 + (g_\sigma(k) - 1) \left\{ 1 + \frac{1}{L} \mid g_E(k) \mid^2 \right\}$$

Thus the instrumental component of the intensity ACF near zero lag has the form

$$\frac{1}{L} \mid g_E(0) \mid^2 g_\sigma(0) \approx \frac{1}{L} \left(1 + \frac{1}{\nu} \right)$$

where ν is the order parameter of the cross-section fluctuations, whereas the total value is

$$\frac{1}{L} \left(1 + \frac{1}{\nu} \right) + \frac{1}{\nu}$$

Let us now demonstrate the complex field correlation properties using SLC data from the same SAR radar as the intensity image above. In this mode, the DLR ESAR has resolution of 4 m in slant range and 0.72 m in azimuth. The sample spacing is 2.5 m in slant range and 0.34 m in azimuth so that this data is over-sampled by 1.6 in slant range and 2.1 in azimuth. Figure 6.14 shows the derived ACF of the complex field. The ENL is 1.0. From Figure 6.14 we note that the width at half height is 1.4 pixels (range) and 1.9 pixels (azimuth), which is consistent with the defined correlations.

Now we construct ACF of the detected intensity image, shown in Figure 6.15. In line with the discussion above, we break the intensity ACF into terms corresponding to those proportional to the correlation of the instrument function, the second and fourth terms above, the flat background (already subtracted) and the third term arising from the incoherent imaging of the cross-section variations. Since the intensity derived from the SLC data has $L \approx 1$ the instrumental profile term can be separated from the total, as shown by the red lines in Figure 6.15. It is apparent that the intensity ACF is consistent with the measured instrumental ACF in Figure 6.14 and the expression above.

The next step is to derive the effective resampling values. The instrumental part of the intensity ACF has a maximum value of 2.2 at zero lag. A fraction of 4% of this would correspond to the 20% of the field ACF, in Figure 6.14. This yields resampling values of 3 and 2 for azimuth and range lines respectively.

This section has demonstrated how the speckle correlations, introduced by the imaging instrument function, appear in the complex field and intensity ACFs. The next stage is to use the information from these ACFs to remove the effects of correlation from the SAR intensity images.

Resampling to remove speckle correlations.

Let us initially consider how to remove speckle correlations when SLC data is available. This is not so in many instances. Therefore it is necessary also to show how the same effect can be achieved when only intensity data is available.

Consider first the instrument ACF shown in Figure 6.14 above. The effect of correlations is insignificant for lag values of 3 pixels in range and 4 in azimuth. If the intensity image were resampled by taking every 3rd pixel in range and 4th in azimuth we would expect the samples to be essentially uncorrelated. This somewhat crude recipe is easy to apply, and has the desired effect. Unfortunately, it is found to lose some of the information contained in the images for two

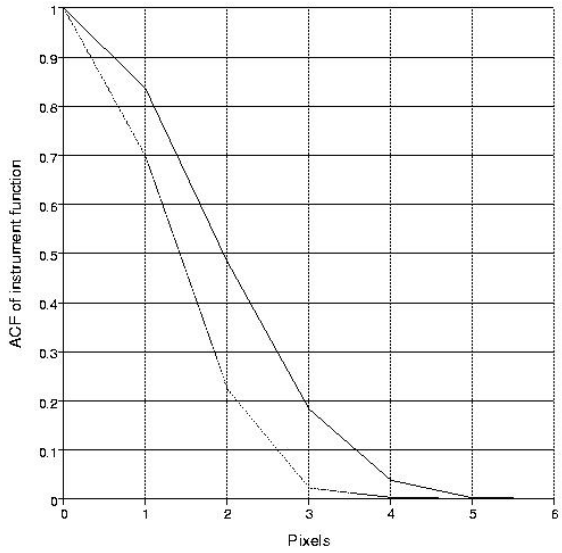

Figure 6.14. ACF of the complex field for the SLC image. Azimuth response (full line), slant range response (dashed line)

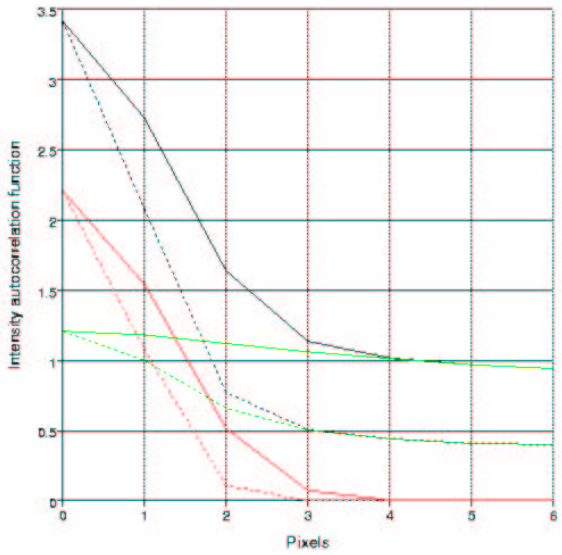

Figure 6.15. Intensity ACF for the SLC data. Azimuth response (full lines), range (dashed). Terms proportional to instrument response (red); cross-section ACF (green).

reasons. Firstly, the actual instrumental correlation lengths are expected to be 1.6 and 2.1 pixels respectively so that further separation of samples is losing uncorrelated information and degrades the resolution. Secondly, if we take averages of the intensity over the resampling interval, the speckle in the resultant image is reduced. The reduction is not equal to the number of pixels averaged, of course, because the speckle is correlated. However, this averaging slightly mitigates against the loss of information resulting from degraded resolution. In practice, we find that taking samples at a spacing corresponding to about 20% correlation along each axis achieves the desired effect. In this example, therefore, the resampling spacing should be 2.1 and 2.9 pixels respectively. Thus resampling corresponds to an average and shrink operation on the intensity over 2 and 3 pixels in range and azimuth respectively.

Let us now investigate how this applies to the SLC imagery.

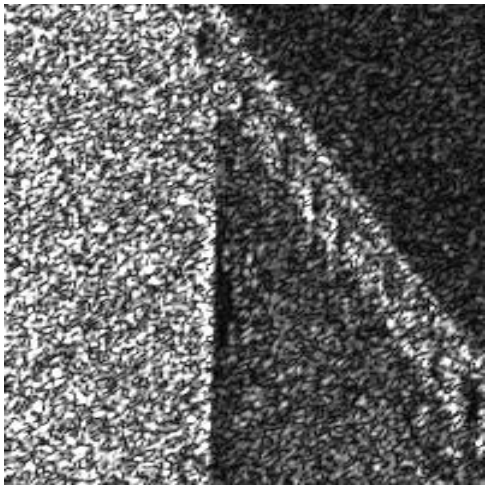

Figure 6.5.3(a)

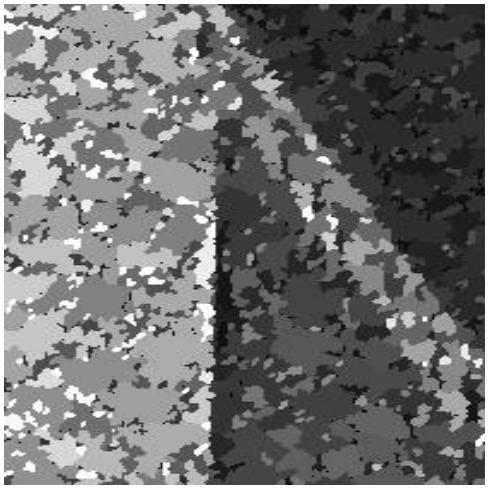

Figure 6.5.3(b)

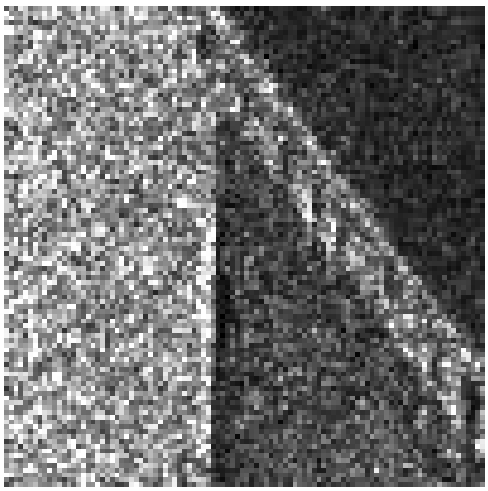

Figure 6.5.3(c)

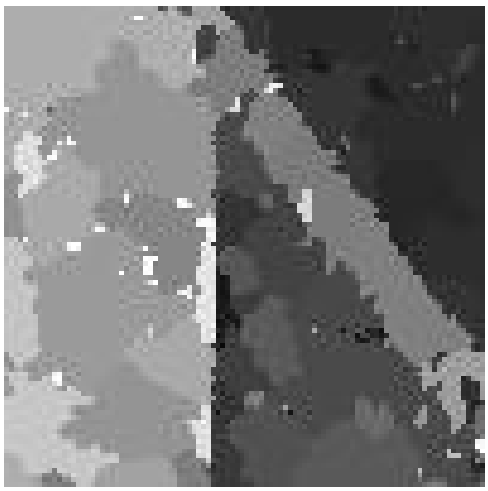

Figure 6.5.3(d)

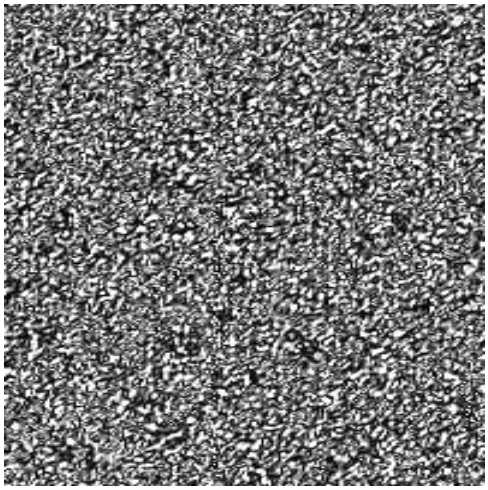

Figure 6.5.3(e)

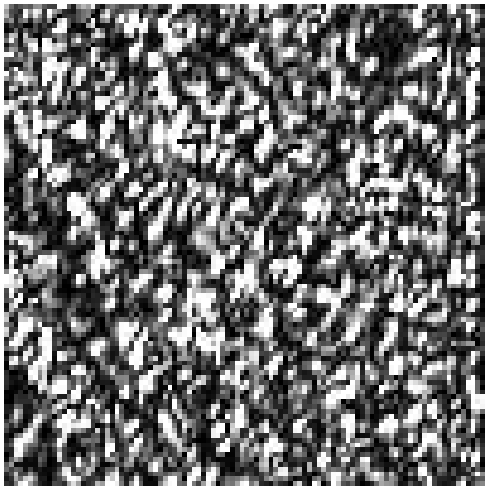

Figure 6.5.3(f)

Figure 6.5.3

Comparison of original and resampled segmentation. (a) original image; (b) segmentation of original image; (c) resampled image; (d) segmentation of resampled image; (e) ratio image for original data; (f) ratio image after resampling.

In Figure 6.5.3(a) we show a detail from the intensity derived from the SLC image. The segmented version, using default parameters, illustrated in Figure 6.5.3(b), again shows the effects of speckle correlations. Many of the regions in this image are clearly resulting from original speckle blobs. Both these images have been rescaled for a square display. Next we apply the resampling derived above; applying average and shrink to the intensity of 2 pixels in range (x) and 3 in azimuth (y). The resampled intensity is shown in Figure 6.5.3(c) with the corresponding segmentation in 6.5.3(d). This segmentation has the expected form with large areas of uniform cross-section and very few speckle-sized segments. The ratio image for the original data (6.5.3a) divided by the segmentation (6.5.3b) is shown in Figure 6.5.3(e). The "speckle" blobs are clearly not of the correct size for the expected correlations, indicating that much of the speckle fluctuation is appearing in the segmented version. The residual is merely the contributions which are uncorrelated on the scale of a pixel. This is confirmed by calculating the ENL, which is 1.70 instead of the theoretical value of 1.0. Figure 6.5.3(f) shows the corresponding ratio after the original intensity data was resampled. Speckle correlations are now on a much larger scale, as expected, and the calculated ENL is 1.03. Thus, this resampling scheme has indeed removed the effect of speckle correlations to a large extent.

Thus we conclude this section by noting that the effects of speckle correlations in the intensity data can be removed by estimating the required resampling from the complex field or intensity ACFs. The former is more accurate; the latter can lead to complications since it includes the ACF of the cross-section.

6.5.4 Resampling applied to the original intensity image

Let us conclude this study by applying the methods discussed in the previous section to the original intensity image. The first step is to construct the intensity ACF of the data, which is shown in Figure 6.16. The situation is complicated by the fact that the cross-section component of the ACF, at lags greater than 4 pixels has an appreciable slope. From the theoretical expression for the intensity ACF we see that the component of the ACF proportional to the instrument function has the form near zero

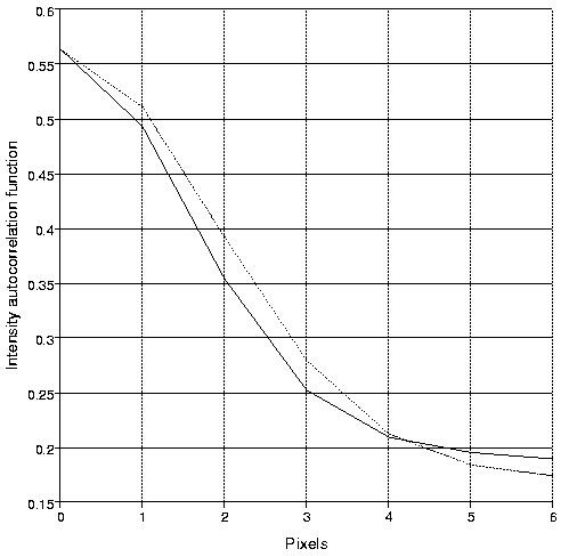

Figure 6.16. Plots of the intensity ACF of the data in Figure 1. Azimuth ACF (full line), range ACF (dashed line).

lag of $\frac{1}{L} \mid g_E(k) \mid^2 g_\sigma(k) \approx \frac{1}{L}\left(1+\frac{1}{\nu}\right)$ where ν is the order parameter of the cross-section fluctuations. The original data has $L \approx 4$.

The origin of Figure 6.16 then indicates that $\nu = 4$ and that the instrument function component of the intensity ACF has a magnitude of 0.31. Thus the correct lag values for resampling are at an ACF value of about 0.26, i.e. lags of 2.9 (azimuth) and 3.3 (range). The nearest integer resampling number is 3 along each axis. It should be noted that it is difficult to separate the instrumental and cross-section components of the ACF in this fashion. Fortunately we only need to approximate the resampling values to the nearest pixel. A reasonable approximation can be obtained by observing that the gradient of both ACFs is reduced significantly at around a lag value of 3, indicating that the speckle part of the ACF has decayed away at about this value. When the averaging and resampling by 3x3 is applied to the original image the image shown in Figure 6.5.4(a) is obtained, while Figure 6.5.4(b) shows the corresponding segmentation. It is obvious that the segmented form of the resampled image contains large uniform regions, as one would expect. This is in stark contrast to the original version in Figure 6.5.2(b). There are no speckle blob sized segments in Figure 6.5.4(b), which should be compared with the very large number visible in 6.5.2(b). Figure 6.5.4(c) shows the ratio of the original intensity to the reconstructed cross-section after resampling. This has the expected scale of correlations in the speckle blobs, similar to those in Figure 6.5.2(d) and completely different from the ratio image obtained from the original data, shown in Figure 6.5.2(c) and repeated here for comparison in Figure 6.5.4(d). Finally, the ENL derived from the ratio image is 4.00, consistent with the defined value. This consistency indicates that the segmentation derived in this way can be trusted.

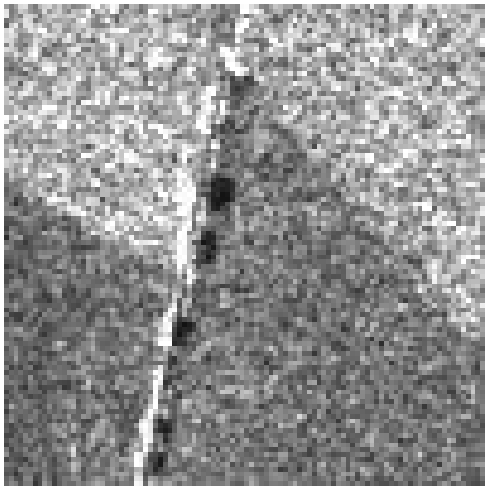

Figure 6.5.4(a)

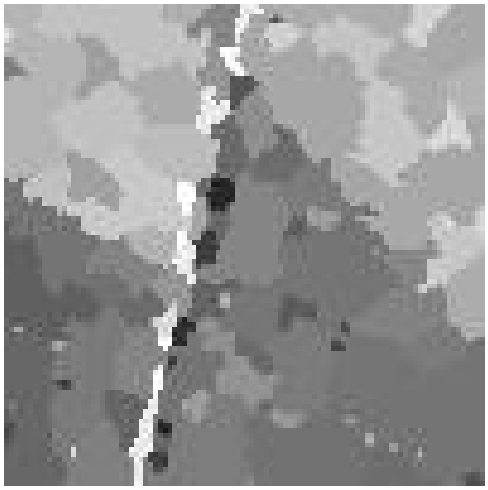

Figure 6.5.4(b)

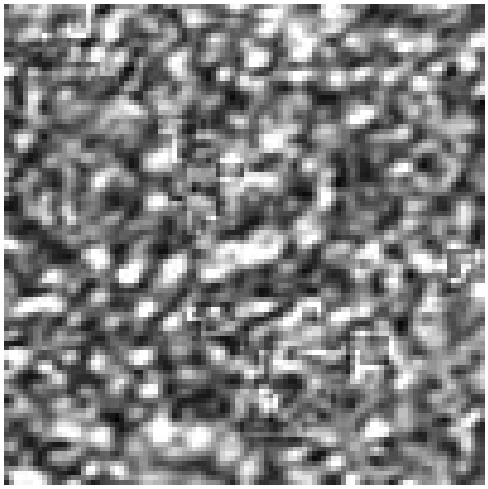

Figure 6.5.4(c)

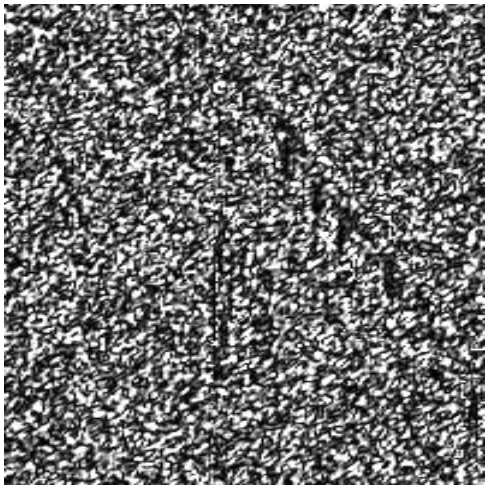

Figure 6.5.4(d)

Segmentation of the resampled original image from Figure 6.5.2. (a) Image resampled by 3x3; (b) segmentation of (a); (c) ratio of (a) to (b); (d) corresponding ratio for original data (from Figure 6.5.2(c)).

6.5.5 Segmentation: Conclusions

In summary, this study has demonstrated the following aspects of applying segmentation to correlated data:

- If segmentation is applied directly to SAR intensity images that have been sampled at spacing less than the speckle correlation width, which is determined by the resolution of the system, the resulting segmentation is dominated by speckle-sized segments.
- The ratio of the original intensity to the segmented version does not correspond to the expected speckle both in apparent ENL and the speckle correlation properties. Only residual correlations on a scale of one pixel remain.

- If the SLC image is available, the speckle correlations can be evaluated directly from the complex field ACF. The resampling factors then correspond to the lag where the field ACF falls to 20% of its value at zero lag.
- Similar information can be derived from the intensity ACF, though more indirectly. The component due to the instrumental response can be separated from that due to the underlying cross-section fluctuations. The resampling factors then correspond to the lags where the instrumental part of the ACF falls to 4% of its value at zero lag.
- Since the resample factor needs only be accurate to the nearest pixel, it is often adequate to identify where the slope of the intensity ACF changes significantly. This marks the point where the instrumental component has largely decayed away.
- Both methods have been applied to the resampling and segmentation of data from both SLC imagery and multi-look intensity imagery. In both cases the residual speckle component is consistent with the expected ENL and the correlation properties similar to those expected.
- These techniques offer the means to segment intensity data with any speckle correlation properties in an optimum fashion; optimising both the retention of information by speckle reduction through averaging the intensity and correlation removal by resampling.

6.6 Correlated Images and Despeckling

Correlations in the data cause the same problems when despeckling as when segmenting. Despeckling uses a neighbourhood of 3x3 pixels to form the reconstruction, which in the example studied in the previous section (6.5.2) is matched to the correlation between samples, in this case. The result of despeckling image 6.13 would therefore be expected to resemble the original image with the individual speckle blobs well reconstructed.

In Figure 6.6(a) we show a detail from Figure 6.13, together with its despeckled version in Figure 6.6(b). The problem cause by the correlations is apparent: the speckle has been well reconstructed!

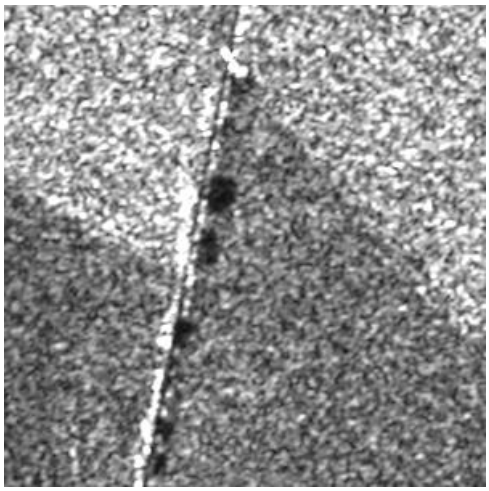

Figure 6.6(a)

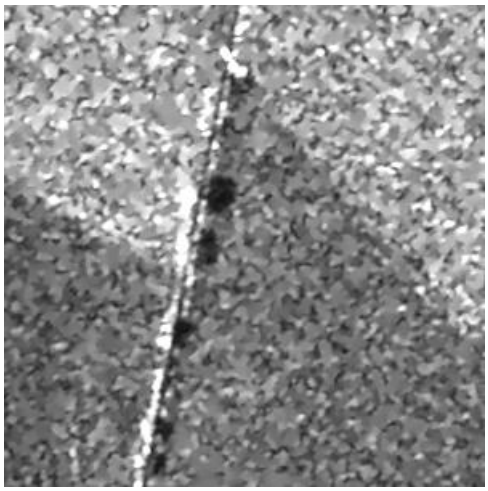

Figure 6.6(b)

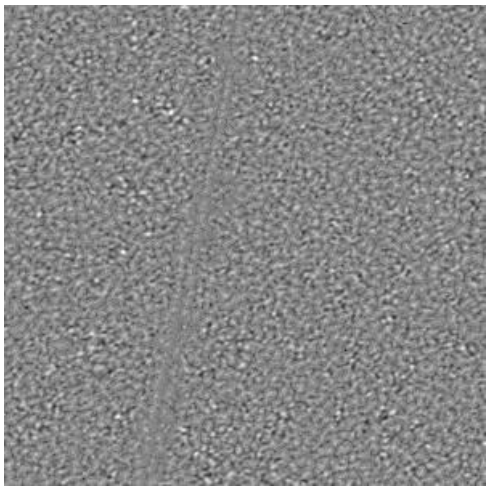

Figure 6.6(c)

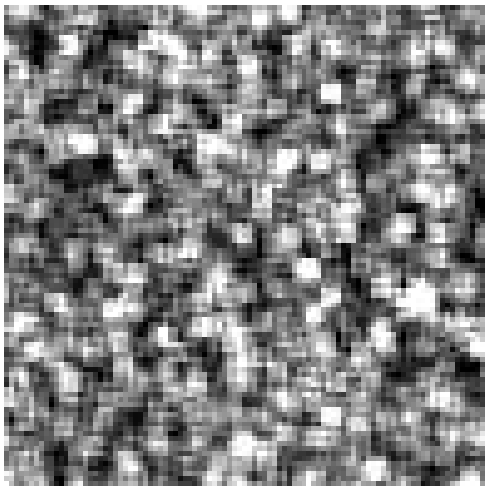

Figure 6.6(d)

Figure 6.6 (a) A detail from the original image shown in Figure 6.13 (b) The image despeckled using `despeckle` (c) The residual speckle (d) The expected residual speckle

If we take the ratio of the original intensity to the despeckled version, the result should have the properties of random speckle with the expected correlation properties and relative variance. In Figure 6.6(c) we show this residual speckle, derived from Figures 6.6(a) and (b). Figure 6.6(d)

comprises a simulated speckle image having approximately the correct correlation properties. It is clear that 6.6(c) fails to show the expected speckle correlation properties, a consequence of the despeckling algorithm tending to reconstruct the individual speckle blobs. This is further borne out by the relative variance of Figure 6.6(c), which is 37.9, compared with the ENL of 4.0.

6.6.1 Resampling to remove speckle correlations

Since the original sampling was at approximately one third of the instrument resolution we would expect following the previous study that speckle correlations could be removed by averaging and shrinking the intensity data by a factor of 3 in each direction. When the averaging and resampling by 3x3 is applied to the original image, the image shown in Figure 6.6.1(a) is obtained, while Figure 6.6.1(b) shows the corresponding despeckled reconstruction. It is obvious that this contains large uniform regions, as one would expect and is in stark contrast to the original version in Figure 6.6(b). There are no speckle blob sized segments in Figure 6.6.1(b), which should be compared with the very large number visible in 6.6(b). Figure 6.6.1(c) shows the ratio of the original intensity to the reconstructed cross-section after resampling. This has the expected scale of correlations in the speckle blobs, similar to those in Figure 6.6(d) and completely different from the ratio image obtained from the original data, shown in Figure 6.6(c) and repeated here for comparison in Figure 6.6.1(d). Finally, the ENL derived from the ratio image is 4.03, consistent with the defined value. This indicates that the despeckling derived in this way can be trusted.

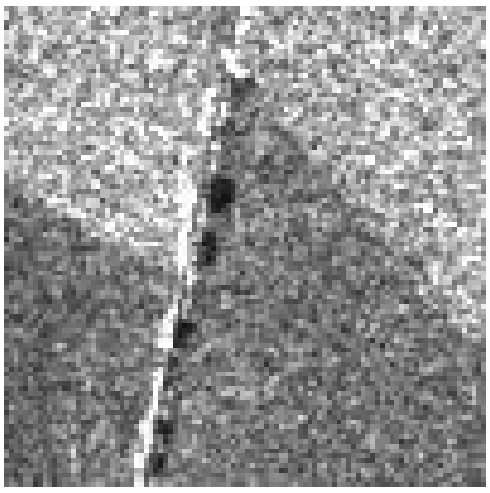

Figure 6.6.1(a)

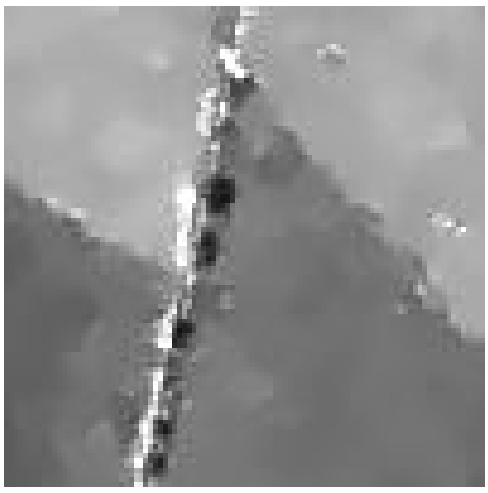

Figure 6.6.1(b)

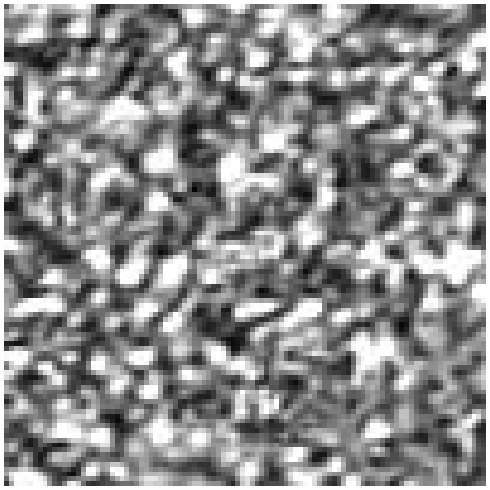

Figure 6.6.1(c)

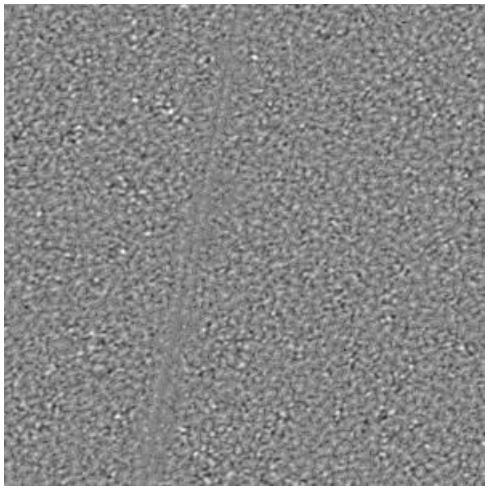

Figure 6.6.1(d)

Figure 6.6.1 Segmentation of the resampled original image from Figure 6.6. (a) Image resampled by 3x3; (b) segmentation of (a); (c) ratio of (a) to (b); (d) corresponding ratio for original data (from Figure 6.6(c)).

6.6.2 Conclusions

This study has demonstrated the following aspects of applying despeckling to correlated data:

- If despeckling is applied directly to SAR intensity images that have been sampled at spacing less than the speckle correlation width, which is determined by the resolution of the system, the speckle blobs are faithfully reconstructed, rather than averaged out.
- The ratio of the original intensity to the despeckled version does not correspond to the expected speckle both in apparent ENL and the speckle correlation properties. Only weak residual correlations on a scale of one pixel remain
- The effects of correlated speckle can be removed by averaging the intensity over an area consistent with the speckle correlation lengths and then subsampling at this spacing, to remove sampling correlations.
- The correlation lengths can be obtained directly from the quoted instrumental resolution and sample spacing. Alternatively it can be derived from the ACF of either the complex (SLC) or intensity images.
- This technique offers the means to despeckle intensity data with any speckle correlation properties in an optimum fashion; optimising both the retention of information by speckle reduction through averaging the intensity and correlation removal by resampling.

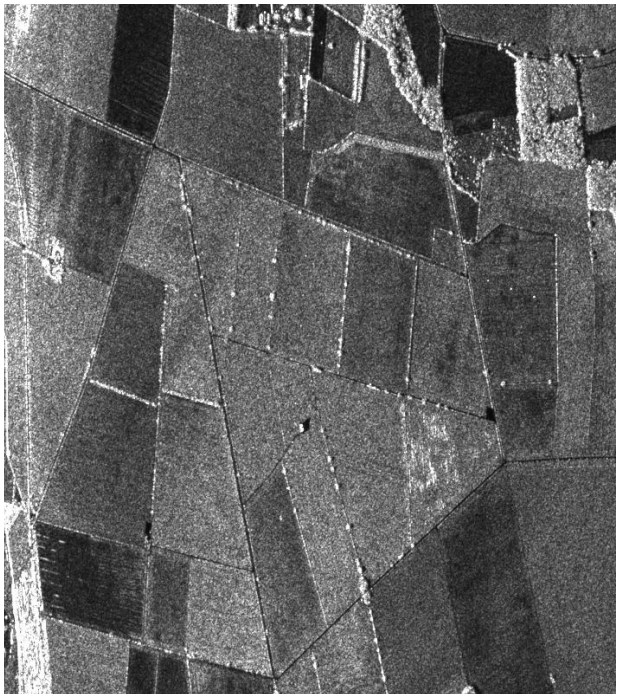

Figure 6.17. An Example DLR ESAR Image

6.7 An Example: Segmenting and Despeckling a Large Image

Figure 6.17 shows an original intensity DLR ESAR image after it has been averaged and resampled 3x3 to remove correlations as discussed above.

We shall first segment this image and then despeckle it. In each case we show the residual ratio to indicate how effective the image-interpretation process is at removing speckle. The quality of reconstruction will be indicated by the absence of structure in the residual image and also the derived ENL. The results motivate our overall advice: despeckling on its own provides smoothed images which retain detail and are easier to interpret by eye than the original data; but for most purposes segmentation provides a better starting point for a processing chain.

6.7.1 Segmentation

Initially we segment the scene using the default `segment` parameter set, except that the shape parameter is set at 0.02. The segmented result is illustrated in Figure 6.18. Most of the structure is clearly visible in the reconstruction. However, some of the fine structure to the right of the top of the runway has not been reproduced. In addition, there are a large number of small regions visible within the scene. These may be partly caused by sloping field areas in the scene, which cannot be presented properly by the uniform region model imposed by segmentation other than by introducing small segments. Alternatively, they might be the result of small statistical fluctuations caused by speckle.

The most revealing test for the quality of the reconstruction is to take the ratio of the original intensity image to its segmentation. If the segmentation corresponded precisely to the underlying RCS, then this ratio would have the properties of pure speckle. The residual ratio would therefore

Figure 6.18. Segmentation of the scene in figure 6.17

show no structure, related to the underlying scene, and the average statistical properties could be used to derive an Effective Number of Looks (ENL) that should be the same as the known value (4.0) imposed by the sensor. This residual ratio image is shown in Figure 6.19. Some minor evidence of the underlying scene structure is visible, indicating that the reconstruction is not perfect. On the other hand, the derived ENL is 4.03, which is close to the expected value. The fact that the ENL is approximately correct indicates that the *average* properties are well represented. However, the presence of structure effects in the residual image indicates those regions where detail has been lost in the reconstruction. This is a consequence of adopting a single shape parameter. If the segmentation algorithm used in `segment` were extended so that this parameter was allowed to adapt to the scene content the effects of structure would probably be reduced (but we have not tried this).

A more sensitive test as to whether the residual ratio corresponds to the predicted speckle model for a system with ENL = 4.0 is to compare the PDF of the residual ratio, at full resolution, with theory. This is illustrated in Figure 6.20. While the agreement is not perfect, it is clear that the PDF has approximately the correct form. The estimated mean for the residual ratio is almost exactly 1.0, as predicted. The observed discrepancy is not, however, entirely due to the fact that the estimated ENL = 4.03, rather than 4.0; there is actually a slight departure from the form of a gamma PDF.

One way to eliminate differences between neighbouring segments that are relatively insignificant is to apply the `merge` algorithm. Rather than providing a global solution to the cartoon model implying uniform regions (segments), this examines pairs of adjoining segments and tests the statistical significance of their separation. This can be expressed in terms of the false alarm probability, P_{fa}, which describes the probability that a single segment could be split into two separate segments by speckle noise. For example, $P_{fa} = 10^{-5}$ would indicate that there was a probability of 10^{-5} that a segment containing a uniform RCS could be split by speckle noise. If a split of this probability is

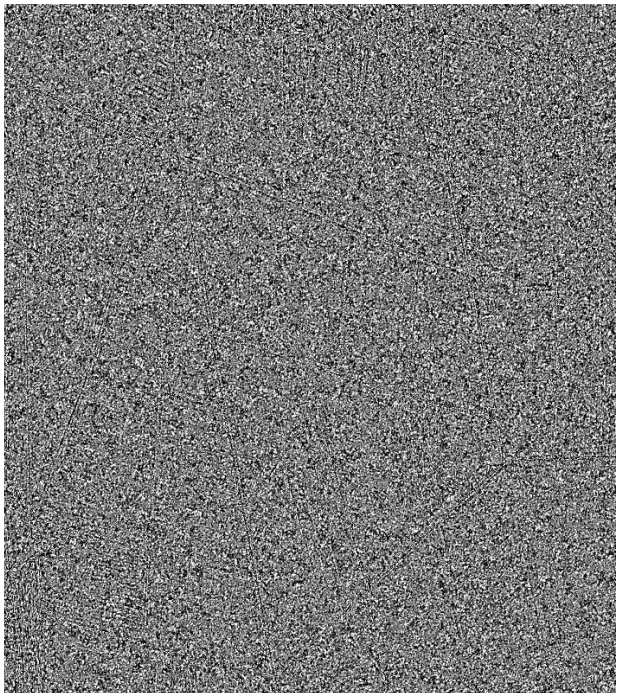

Figure 6.19. Residual ratio derived from figures 6.17 and 6.18

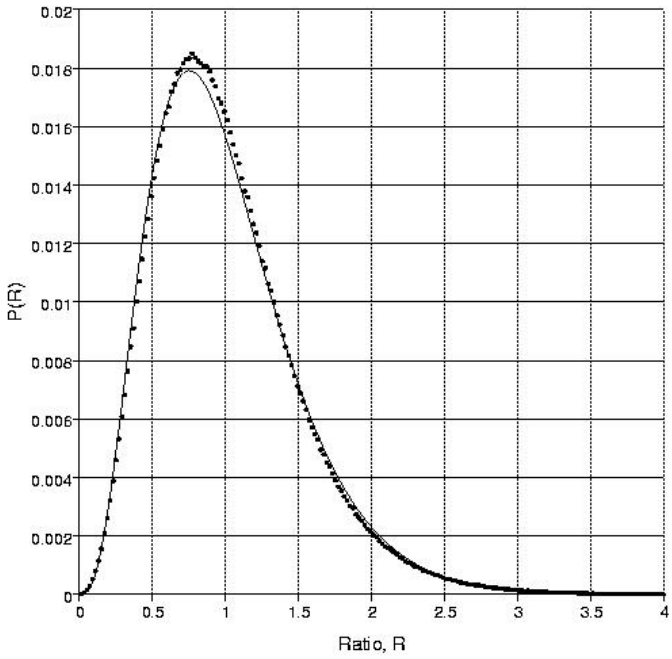

Figure 6.20. Comparison of observed residual ratio (dotted curve) for segmented data, shown in Figure 6.19, with theoretical PDF for ENL = 4.0 (full curve) having mean = 1.0 and order = 4.0.

Figure 6.21. Result of merging segments with $P_{fa} = 10^{-5}$

not significant then such regions should be merged; and this is what `merge` does.

The result of applying this merge process to the data of Figure 6.17 and the segmentation of Figure 6.18 is shown in Figure 6.21. There is no appreciable difference between this and the original segmentation in Figure 6.18, indicating that the many small regions visible are very unlikely to be a consequence of speckle fluctuations, and hence are indeed significant within the cartoon segmentation model used.

6.7.2 Despeckling

Another method for removing speckle is to use the despeckling algorithm implemented in `despeckle`. Rather than impose the cartoon model of uniform RCS within each segment, this implies a local similarity over a neighbourhood of 3x3 pixels. It identifies which possible arrangement of 3 correlated pixels, out of the 12 possible, is most consistent with the data and speckle. It too depends on simulated annealing to determine the overall optimum despeckled solution. The consequence of applying this algorithm to the data of Figure 6.17 is shown in Figure 6.22. Since `despeckle` does not impose uniform regions with consequent sharp discontinuities (edges), the result appears to yield a gently varying background. This is likely to be more consistent with the actual real data than the cartoon model. One might deduce that this would be a better algorithm to use for image interpretation. However, the main weakness of this method is that it merely generates a continuously varying intensity, largely free of speckle noise, which still requires further image processing to identify regions of similar properties, i.e. classification. Segmentation can lead directly into this stage. Furthermore, there is no concept of an edge in this approach, whereas most information data bases, such as maps, require regions of similar content to be identified and their edges delineated.

The final drawback of this approach is that it actually misrepresents the speckle model. If we

Figure 6.22. Despeckled version of Figure 6.17.

construct the residual ratio, assuming that the despeckled output corresponds to the underlying RCS, we obtain the result shown in Figure 6.23.

While the uniform regions of the original image (Figure 6.17) yield a realistic speckle appearance, all structure in the image affects the residual ratio strongly. This is a consequence of the 3×3 window over which correlations in the data are identifed and exploited in the reconstruction process. When this is compared with the corresponding result following segmentation, in Figure 6.19, it is clear that the process introduces considerable distortion into the reconstruction. This is made apparent when the ENL is estimated, yielding a value of 4.26, compared with 4.03 for segmentation. This indicates the extent to which this method fails to satisfy the speckle model.

The effect is made still more obvious if the PDF of the residual ratio in Figure 6.23 is compared with the predicted gamma PDF of mean 1.0 and order 4.0. This is illustrated in Figure 6.24. It shows that both the order of the distribution and its shape differ from the ideal theory.

6.7.3 Conclusions

- We have illustrated the effect of removing speckle by invoking two different models: the cartoon model for segmentation and the local similarity model for despeckling.

- Segmentation yields the result that is closest to the speckle model in that the residual ratio resembles speckle most closely.

- Segmentation shows evidence of many small segments. These are retained when the data is merged, indicating that they are statistically significant.

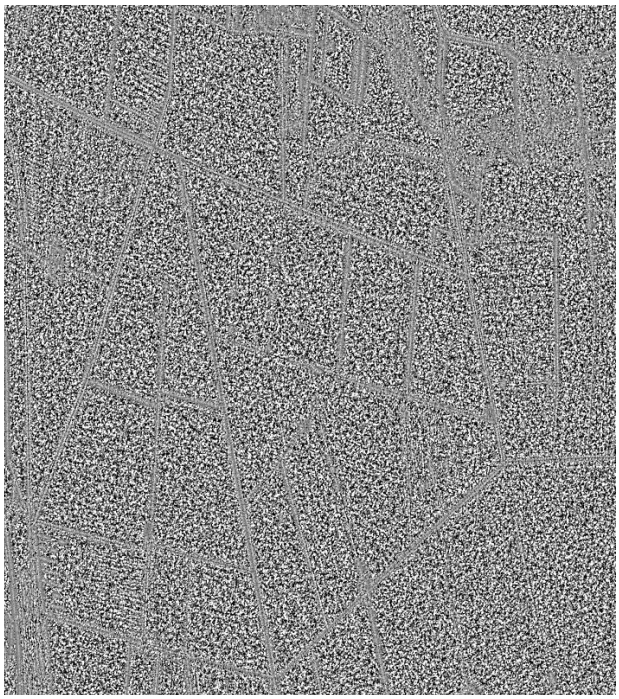

Figure 6.23. Residual ratio for the despeckled reconstruction in Figure 6.22.

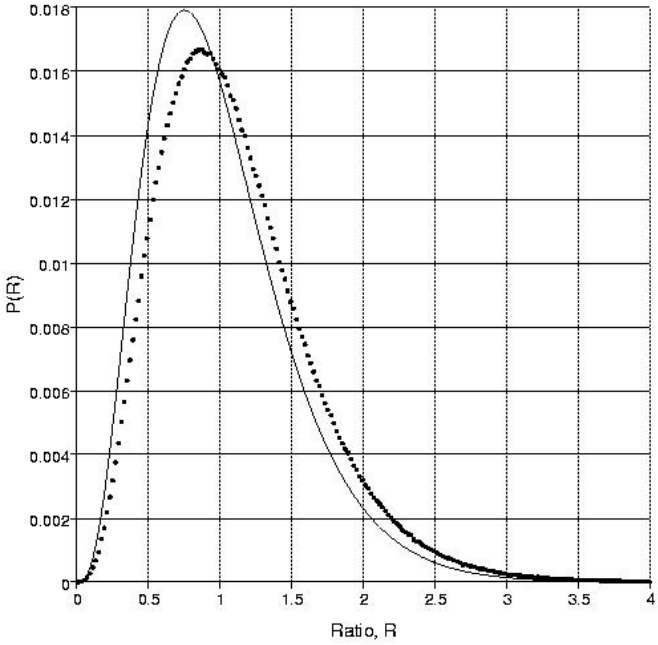

Figure 6.24. Comparison of observed residual ratio for despeckled data, shown in Figure 6.22, with the theoretical PDF for ENL = 4.0 (full curve) having mean = 1.0 and order = 4.0.

- Despeckling presents a more continuous fluctuation in the background, inherent in the local similarity model.
- Despeckling shows strong correlations around structures in the scene as a consequence of the 3x3 construction window.
- Despeckling yields a poorer fit to the speckle model.

6.8 Segmentation and Classification of Polarimetric Data

6.8.1 Introduction

In many remote sensing applications important additional information can be derived from multipolarised imagery. The different polarisations respond in different ways to the orientation and shape of the objects from which scattering takes place. It is obviously important to have tools that make full use of this information, and in this Case Study we look at the steps needed. As previously, we simulate data of the required type, in this case HH, HV and VV complex imagery. The underlying class structure in the simulated image is shown in Figure 6.25(a). The specified ground truth is illustrated in Figure 6.25(b) while the training and test masks are shown in Figures 6.25(c) and (d) respectively.

In order to simulate polarised imagery we specify the class covariance matrix for each of the six classes shown in Figure 6.25. The covariance matrix has the form:

$$\begin{vmatrix} E_{HH}E_{HH}^* & E_{HH}E_{HV}^* & E_{HH}E_{VV}^* \\ E_{HV}E_{HH}^* & E_{HV}E_{HV}^* & E_{HV}E_{VV}^* \\ E_{VV}E_{HH}^* & E_{VV}E_{HV}^* & E_{VV}E_{VV}^* \end{vmatrix}$$

The values of the elements for each class are listed in Table 6.8.1. Note that the element (0,0) is identical for each class. This means that the HH image will show no structure.

1:	1+0j	0j	0j
	0j	1+0j	0j
	0j	0j	0.1+0j
2:	1+0j	0.68+0j	0.21+0j
	0.68+0j	1.8+0j	0.12+0j
	0.21+0j	0.12+0.01j	0.15+0j
3:	1+0j	0.43-0.06j	0.1-0.2j
	0.43+0.06j	1.2+0j	0.4-0.01j
	0.1+0.2j	0.4+0.01j	0.18+0j
4:	1+0j	0.73+0.73j	0.092+0.046j
	0.73-0.73j	2.1+0j	0.23+0.15j
	0.092-0.046j	0.23-0.15j	0.08+0j
5:	1+0j	1.2-0.2j	0.05-0.05j
	1.2+0.2j	2.6+0j	0.05+0.1j
	0.05+0.05j	0.05-0.1j	0.11+0j
6:	1+0j	0.3-0.6j	0.1+0.2j
	0.3+0.6j	1.5+0j	0.09+0.02j
	0.1-0.2j	0.09-0.02j	0.14+0j

Table 6.8.1: Values of the covariance matrix elements for the 6 classes.

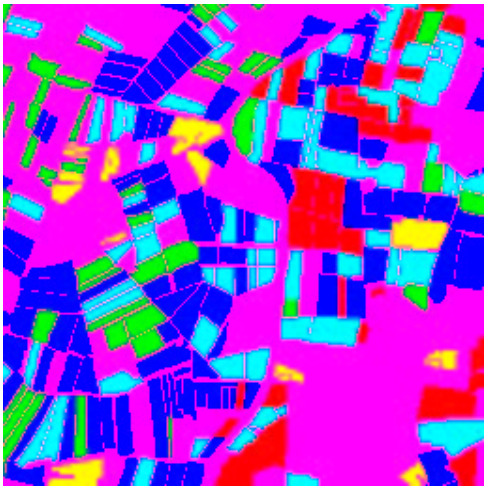

Figure 6.25(a)

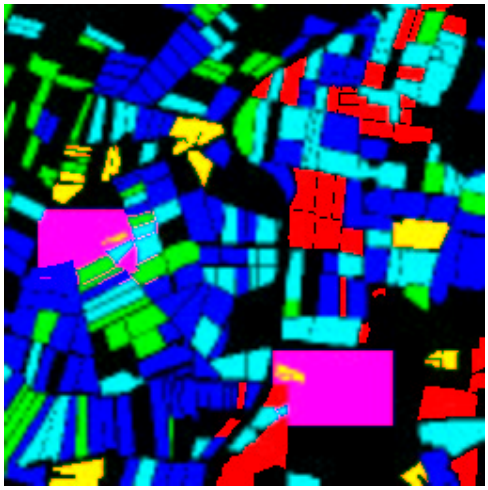

Figure 6.25(b)

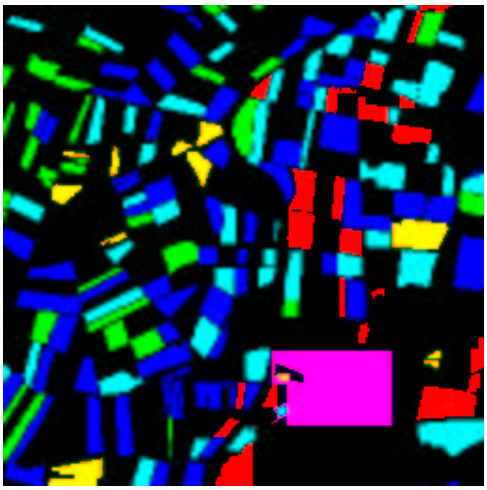

Figure 6.25(c)

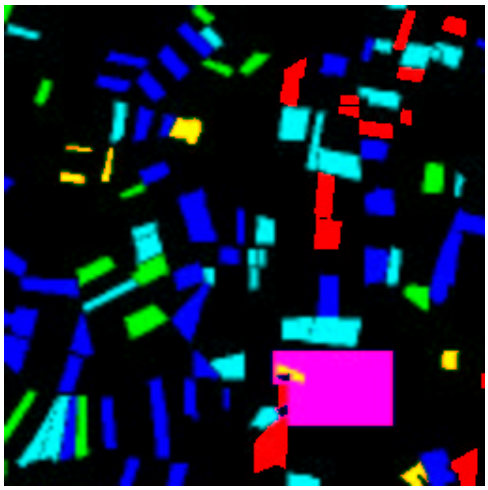

Figure 6.25(d)

Figure 6.25. (a) Simulation classes, (b) ground truth, (c) training mask, (d) test mask. The classes are denoted by 0 (magenta), 1 (blue), 2 (cyan), 3 (green), 4 (yellow) and 5 (red).

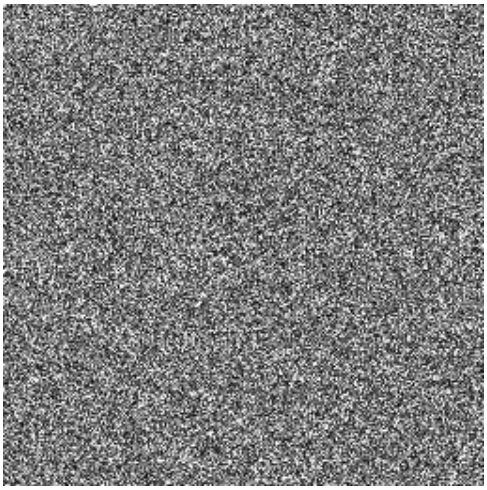

Figure 6.26(a)

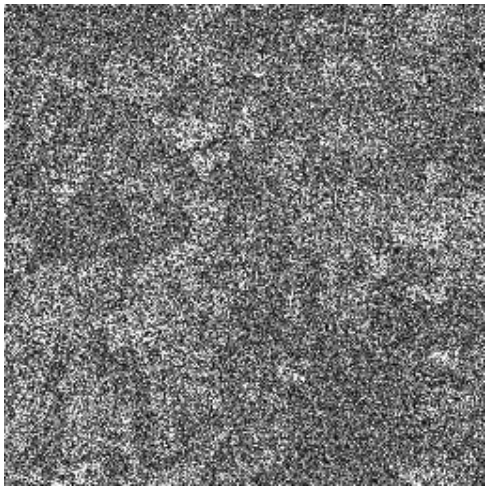

Figure 6.26(b)

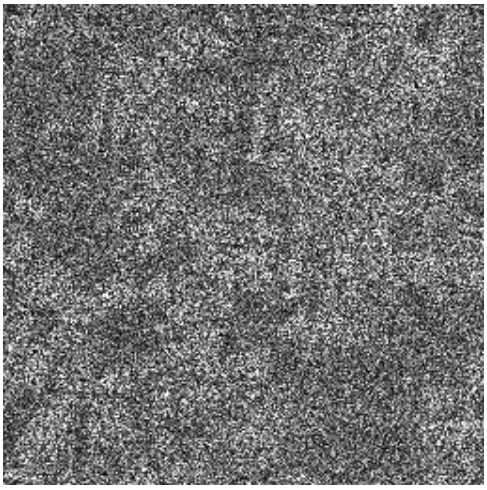

Figure 6.26(c)

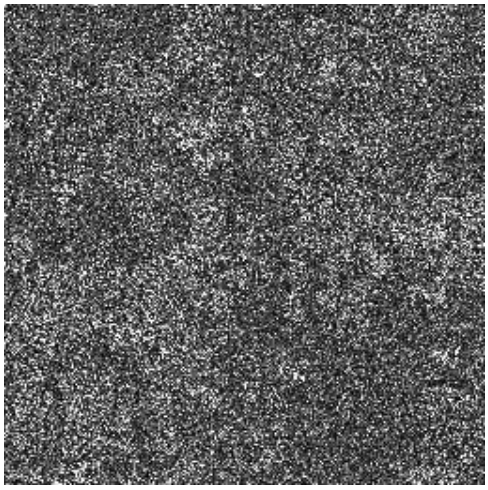

Figure 6.26(d)

Figure 6.26. Amplitude of simulated HH (a), HV (b) VV (c) and span (d) images

An example of simulated imagery for the HH, HV and VV intensities is shown in Figure 6.26. The HH image in Figure 6.26(a) shows no structure, as expected. There is some evidence of structure, however, in the HV and VV images, in Figures 6.26(b) and (c) respectively. Since there is visible structure, we might expect that these images could be segmented jointly in order to determine the structure within the scene. We shall show later that the sensitivity of this measure is poor when compared with full polarised segmentation. The sum of these three images represent the span measure, shown in Figure 6.26(d), which has often been adopted in the analysis of polarised images. It too will be shown to result in poor classification performance.

With the exception of the three individual images or the span measure, shown above, other measures for polarised segmentation are all based on the covariance matrix of the image. This is constructed, as shown above, using data from the three complex simulated images, E_{HH}, E_{HV}, E_{VV}. The covariance matrix carries all the information about the polarised image. Therefore it is

important to process the full complex data for each element. This is implemented in the Maximum Likelihood classifier and the polarised segmentation algorithm, which will be described later. This ML polarimetric classifier takes averages of the polarisation covariance matrix over regions, which ideally correspond to areas of uniform polarisation properties. Various algorithms are used to derive this region map. We either:

- treat the single pixel covariance matrix directly
- apply a 3x3 window to the covariance matrix.
- perform polarised segmentation on the data
- calculate the Upper Limit by averaging the covariance matrix over the original ground truth mask.

Two different forms of polarised segmentation will be compared:

- ML polarimetric segmentation
- segmentation of the span measure.

The first method provides the Maximum Likelihood segmentation solution. The latter is included to demonstrate how much poorer is segmentation which only uses incoherent information, namely the intensity in each polarisation.

The ML polarimetric segmentation algorithm requires the operating parameters to be tuned for "best" performance. As we found previously, segmentation is controlled by three basic parameters:

1. The shape parameter attaches a penalty to excessive curvature in the reconstruction. Large values of the parameter allow many changes of direction of the edge. Small values bias the reconstruction towards smoother edges. Since speckle yields very crinkled edges to regions of similar intensity it is important to adopt a large curvature penalty, i.e. a smaller shape parameter.

2. The annealing method has two further parameters attached. The first is the stopping parameter to assess when the reconstruction is complete. For noisier data one would expect this parameter to be increased. Smoother data, such as that encountered on averaging speckle, should allow a reduced value of the stopping parameter.

3. The second parameter in the annealing method is the cooling parameter which controls the rate at which the annealing converges on the final solution. If this is made too large the solution may freeze out away from the expected result.

It is important to derive values for these parameters that yield the"best" segmentation. Since we have started with simulated test data we can tune the parameters in terms of the final classification performance. This avoids somewhat arbitrary decisions about what constitutes a"good" segmentation.

6.8.2 Segmentation performance

A crucial issue with applying segmentation is the selection of operating parameters. In Figure 6.27 we show the result of applying ML polarimetric segmentation with different values of the shape parameter. In Figure 6.27(a) a shape parameter of 0.1 is adopted. It is apparent that this allows small regions with a sharply-varying boundary to be reconstructed However, there is some evidence for flecks of speckle-dominated segments. By increasing the penalty for non-smooth boundaries (reducing the shape parameter) this type of reconstruction can be suppressed. Figures 6.27(b) to (d) show the corresponding results for shape parameters of 0.01, 0.001 and 0.0001. If you compare these with the original test pattern in Figure 6.25(a) it is clear that reconstruction is generally very good. The shape of the fields around the water course are well represented. Figure 6.27(b), with a shape parameter of 0.01 appears to offer a good compromise between smooth edges to regions and reproducing small structures. We need to increase the shape parameter to follow weak small structure but, on the other hand, we need to reduce the shape parameter to eliminate speckle regions. As before, the only reasonable test is to compare the classification probability, as will be discussed below.

It is instructive to compare ML segmentation results with those derived using the span measure, shown in Figure 6.28. Figure 6.28(a) shows the ML segmentation using the "normal" parameters, discussed later. When the segment boundaries are overlaid onto the original classes, in Figure 6.28(b), two things are immediately obvious. Firstly, the segment boundaries are closely aligned with the region boundaries. This means that there is little blurring of the properties of the different classes. Secondly, quite large segments are reconstructed. Often a complete region of the original scene is represented by a single segment. Even where large homogeneous regions are broken up in segmentation, the individual segments are sufficiently large that the statistical properties will be well determined. On the other hand, there is no appreciable evidence of the field structure in the segmented span image, in Figure 6.28(c), revealing that there is very little information carried by the span measure. Indeed, the overlay of the segmentation boundaries onto the original classes, illustrated in Figure 6.28(d), confirm the inadequacy of the span measure in this context.

The Lower Limit corresponds to attempting ML polarimetric classification on individual pixels. There is no reduction in the statistical uncertainty so that speckle noise would be expected to dominate. Because of this, much research has depended on taking 3x3 averages of the covariance matrix before classification. This reduces statistical uncertainty at the expense of blurring region boundaries. Thus the impact of such a process is scene dependent.

The purpose of the comparison is to reveal:

1. the importance of using the full complex polarimetric data.

2. the considerable advantage of segmentation methods over the 3x3 window and the Lower Limit.

6.8.3 Comparison of classification methods

Optimum classification of polarised images is based on the covariance matrix, which contains all the information in the scattering process. The ML classifier can be defined in terms of the likelihood

$$\lambda = -ln|\mathbf{R}_c| - Trace\lfloor \mathbf{R}_c^{-1}\hat{\mathbf{R}} \rfloor$$

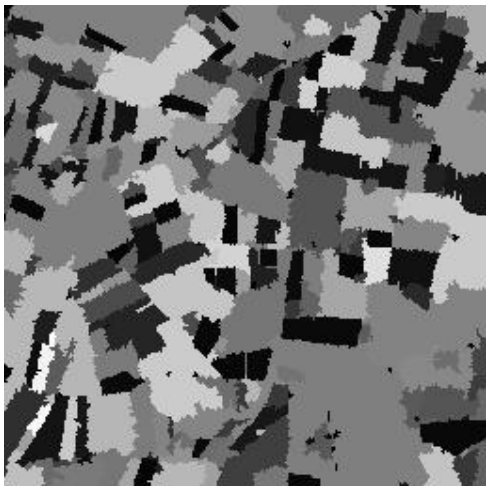

Figure 6.27(a)

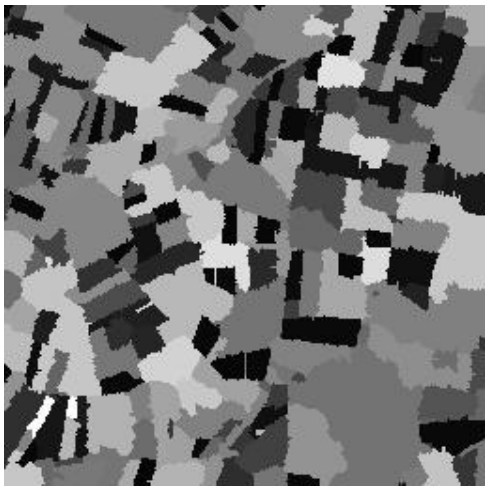

Figure 6.27(b)

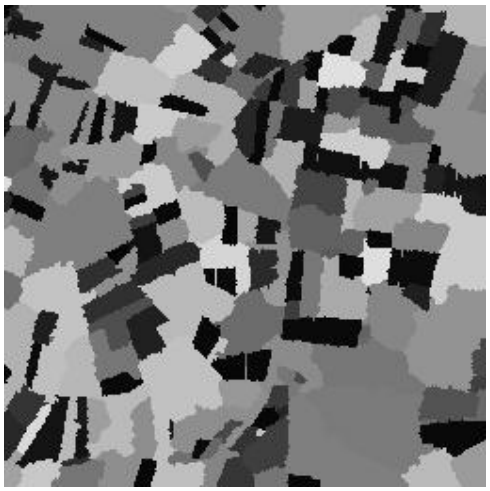

Figure 6.27(c)

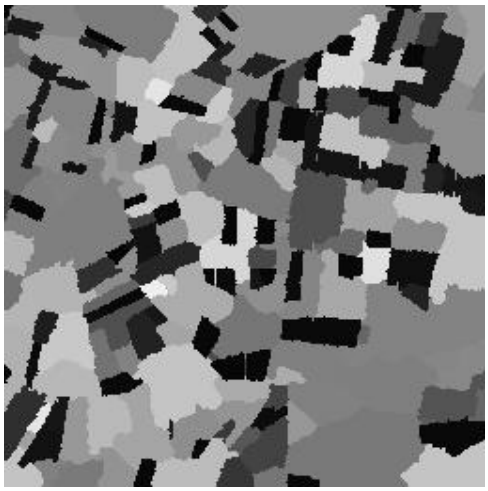

Figure 6.27(d)

Figure 6.27. Segmentation of a single polarised image using different shape parameters. (a) 0.1, (b) 0.01, (c) 0.001, (d) 0.0001.

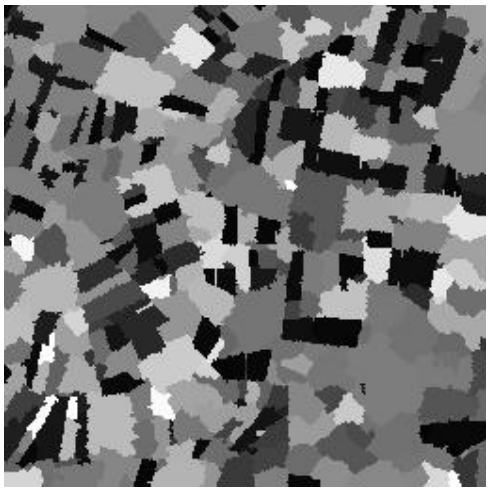

Figure 6.28(a)

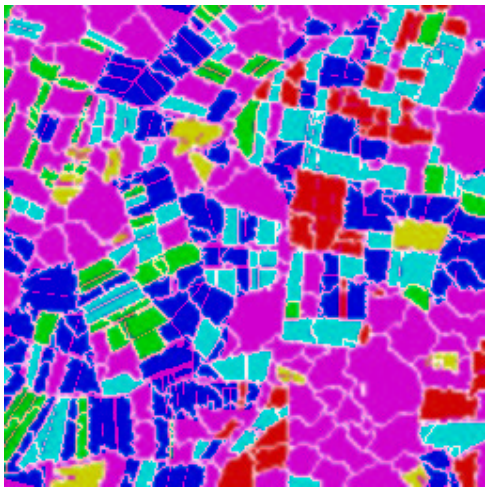

Figure 6.28(b)

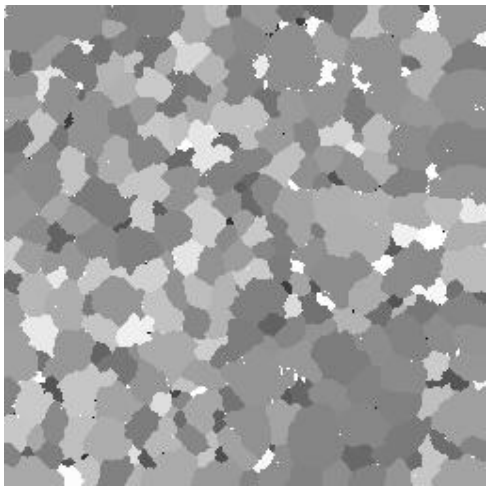

Figure 6.28(c)

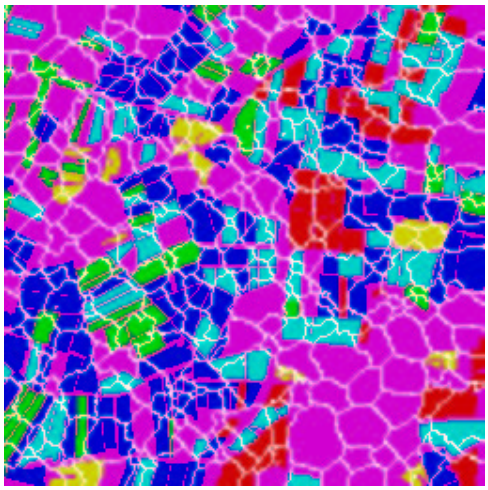

Figure 6.28(d)

Figure 6.28. (a) ML segmentation (b) Overlay of the segmentation boundaries on the original classes (c) Segmentation of the span measure; (d) overlay of the span segment boundaries on the original classes.

where $\mathbf{R}_c$ is the reference covariance matrix for class C and $|\mathbf{R}c|$ denotes its determinant and $\hat{\mathbf{R}}_c$ denotes the estimated covariance matrix from the data. The maximum value of the likelihood, when all possible classes are compared, indicates which class the particular regions should be assigned to.

We now compare classification of the covariance matrix using the following methods:

1. Upper Limit – classifying the average in each ground truth region
2. Segmentation – classifying the average in each segment
3. 3x3 window – classifying the average over a 3x3 window
4. Lower Limit – classifying the covariance matrix in each pixel.

The classification results are shown in Figure 6.29. Classification of a single pixel, shown in Figure 6.29(a) is comparatively poor, with the average probability of correctly classifying each category 0.615. The associated confusion matrix, shown in Table 6.1 (a) reveals how poorly the classes 0 (magenta), 1 (blue) and 4 (yellow) are assigned, as can be seen in Figure 6.29(a). Nevertheless, the underlying structure of the original classes in Figure 6.25(a) is still visible. Classification over a 3x3 window, shown in Figure 6.29(b) yields considerable improvement with the average fraction of correctly classified pixels in each class rising to 0.798. However, the confusion matrix shows that there is still ambiguity between class 2 (cyan) and 0 (magenta) and between class 1 (blue) and class 4 (yellow).

The results of classification following ML polarised segmentation depend on the selection of control parameters. Three basic default sets are defined, sensitive, normal and fast. The “sensitive” parameters should be employed where there is little scene structure, or when it is on a large scale so that large segments are to be preferred. The “normal” parameters are for general use, representing a compromise between sensitivity and speed. Finally, the “fast” parameters are intended for speedy, but not necessarily accurate determination of segments. Note that when we are concerned with classification after a segmentation stage it is not essential that the segmentation should completely fill the homogeneous regions. Provided that segments are large enough for the statistical uncertainty to be small and that the segments do not go over region boundaries the results will be indistinguishable. Indeed, there is no appreciable difference between the classification performance for the three modes, in this example. For the present comparison, we adopt the “normal” mode. The classified results, shown in Figure 6.29(c) is much closer to the ground truth, with the average fraction of correctly classified categories rising to 0.941 and the confusion matrix, in Table 6.1(c) showing very little ambiguity between classes. There is an obvious over-smoothing of some of the boundaries so that details of the structure have been lost, at the expense of rejecting unwanted flecks in the reconstruction. For comparison we show the corresponding result for ML classification following segmentation of the span measure, in Figure 6.29(d). As already noted, there is little correspondence between the segment boundaries and the original scene structure. This mixes the properties of the different classes and results in very poor classification. The average fraction of correctly classified categories is only 0.373, which is worse than all the others. Obviously the span measure is not worth pursuing.

Finally, the Upper Limit, provided by the ground truth itself, leads to the results shown in Figure 6.29(e). The average probability of correct classification is now 0.996, with only a few pixels in class 0 (magenta) incorrectly assigned. Indeed, the confusion matrix shown in Table 6.1(d) reveals very little ambiguity.

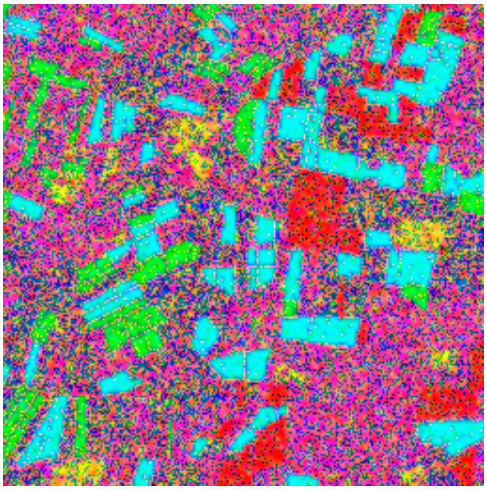

Figure 6.29(a)

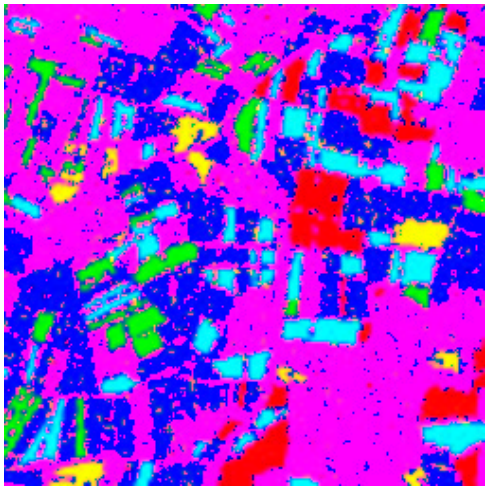

Figure 6.29(b)

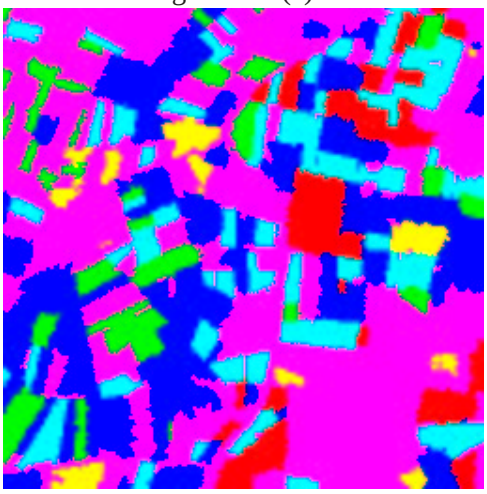

Figure 6.29(c)

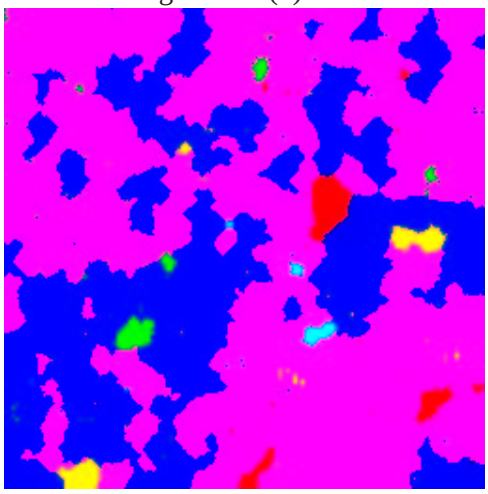

Figure 6.29(d)

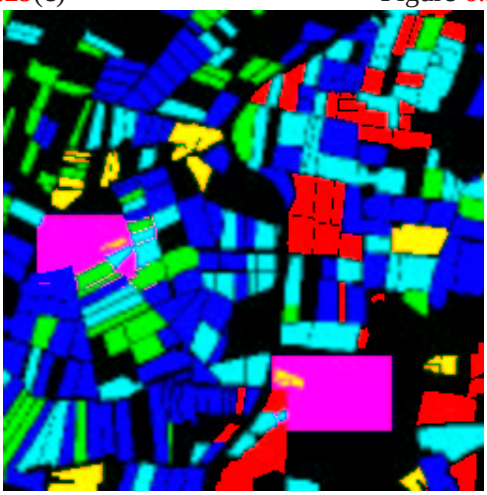

Figure 6.29(e)

Figure 6.29. Classification based on (a) single pixel, (b) 3x3 window, (c) ML polarimetric segmentation, (d) span segmentation, (e) Upper Limit.

	Assigned class					
True Class	0	1	2	3	4	5
0	0.4087	0.0946	0.0662	0.1108	0.1345	0.1851
1	0.1180	0.3069	0.0517	0.1300	0.2070	0.1865
2	0.0119	0.0039	0.9106	0.0628	0.0053	0.0055
3	0.0274	0.0209	0.1234	0.7596	0.0431	0.0255
4	0.0705	0.0886	0.0579	0.1284	0.5533	0.1013
5	0.06663	0.0371	0.0397	0.0522	0.0527	0.7520

(a) Single Pixel

	Assigned class					
True Class	0	1	2	3	4	5
0	0.8858	0.0858	0.0034	0.0101	0.0149	0
1	0.0995	0.8687	0	0.0002	0.0283	0.0033
2	0.1902	0.0959	0.6801	0.0053	0.0284	0
3	0.1135	0.1095	0	0.7690	0.0080	0
4	0.0713	0.1846	0	0.0037	0.7367	0.0037
5	0.1169	0.0384	0	0	0	0.8447

(b) 3x3 window

	Assigned class					
True Class	0	1	2	3	4	5
0	0.9108	0.0358	0.0149	0.0365	0.0020	0
1	0.0646	0.9263	0.0009	0.0015	0.0067	0.033
2	0.0445	0.0003	0.9552	0.0003	0	0
3	0.0105	0.0059	0.0026	0.9811	0	0
4	0.1049	0.0018	0	0.0054	0.8879	0
5	0.0094	0.0021	0.0052	0	0	0.9833

(c) Segmentation

	Assigned class					
True Class	0	1	2	3	4	5
0	0.9757	0.0101	0.0014	0.0014	0.0055	0.0061
1	0	1.0000	0	0	0	0
2	0	0	1.0000	0	0	0
3	0	0	0	1.0000	0	0
4	0	0	0	0	1.0000	0
5	0	0	0	0	0	1.0008

(d) Upper Limit

Table 6.1. Confusion matrix for the different classification methods. The fraction of the true class assigned to each category is given

6.8.4 Conclusion

- It is apparent from this comparison that it is pointless attempting classification on individual polarised pixels in SAR.
- Averaging over a 3x3 window improves the result. The accuracy could be further improved by increasing the size of the window. However, there is then a problem of averaging out small regions. What is required is a method that adapts the region size to the data; this is segmentation.
- Segmentation followed by classification yields a considerable improvement, since it provides a filter whose window size and shape is adapted to the data. The fact that the segments are not matched to the ground truth regions is a consequence of speckle noise. In this instance we know that the simulation was for uniform properties within each region. The speckle statistics have spoilt this. Larger regions could be grown, by changing the curvature parameter. However, this would tend to reject small regions.
- Only the full ML polarised segmentation method offers a significant improvement over the 3x3 window. It reduces the false classifications from about 20% to about 6%.
- The Upper Limit, as one would expect, provides the highest statistical accuracy. There are three important caveats to note, however.
    1. The result is based on the assumption that the properties within the region are indeed homogeneous. In real applications this will not generally be the case.
    2. The Upper Limit is incapable of showing if there are inhomogeneities within the defined reference region. The consequence of any such variation is a degradation in classification performance.
    3. The classification can only be applied where region boundaries are known. It cannot extrapolate beyond the ground truth. Segmentation, on the other hand, can use the training set to enable it to classify over as large an image as is required.

Thus, the overall conclusion is that segmentation provides an essential precursor to polarised classification.

Appendix A

InfoPACK High Level Routine Documentation

In this chapter we provide full specification for all the major (high level) *InfoPACK* routines.

A.1 User Interface

A.1.1 Scripts and Functions

As described in section 1.1, *InfoPACK* routines can be accessed as Python functions, Python scripts or via the GUI. The latter is simply a front end for the Python scripts. The distinction between the two fundamental ways of calling the routines is as follows:

Python Functions: the routine is called as a Python function. The functions are contained in the Python module *infopack* and have a user interface designed to be as convenient as possible to use from within the InfoPACK or Python console. Operations on images are, in general, carried out in memory and so do not provide a suitable environment for handling arbitrarily large images.

Python Scripts (`.py` files). These are designed for users who prefer to use a command line interface with or without Unix shell scripts/Windows batch files. Python scripts work file to file and are memory optimised in order to handle arbitrarily large images efficiently.

The scripts have a very similar syntax to the function form, so that switching between them (should you need to do this) is straightforward. Here is an example:

Command line call: `segment.py image_file output_file quality=sensitive`

InfoPACK Python function: `array_out = segment(array_in, quality=sensitive)`

In the latter call, `array_in` is the previously loaded image array; the result will be stored in the Numeric array `array_out`. Section A.1.3 discusses file I/O for InfoPACK functions in greater detail.

A.1.2 GIS Information

InfoPACK import facilities preserve any GIS information in the original file, and the imported *NetCDF* files can be loaded, processed and saved while continuing to preserve the information. Alternatively, you can ignore it totally if it is not relevant to your application. The way in which this is done is outlined in the next section (Section A.1.3).

A.1.3 File I/O

File I/O is handled behind the scenes when using the Python Script versions of the routines. Python Function calls, however, work via calls to and from the Python Numeric arrays associated with each image file.

Loading and saving of *NetCDF* files is done by calling the utilities `load, save`. The load function returns a list with two elements in it; the first is an array containing the image (in Numeric Python format) and the second is a GIS object. For example: from the console

```
a, gis = load("file.img")
```

returns a list containing a Numeric array (the actual image data) and a class with the GIS information. Any format recognised by the GDAL library is valid.

The following code segments the Numeric array "a", normalise the original image by the intensity value in each segment and saves the resulting file while preserving the GIS information:

```
from infopack import *
a, gis = load("file.img")
b = segment(a)
c = a / b
save(c, "result.nc", gis=gis)
```

Here, `a,gis` is a list with `a` containing the image and `gis` containing the GIS information. All the *InfoPACK* `.py` scripts, and the GUI buttons, preserve GIS information in this way.

However if you do not care about keeping (or using) the GIS information then you can throw it away:

```
from infopack import *
a, dummy = load("file.img")
b = segment(a)
c = a / b
save(c, "result.nc")
```

If you miss out the `", dummy"` then `"a"` will be a list and you will get all sorts of interesting bugs when you try to use `a` as an image (unless you access the numeric array and gis information via list elements `a[0]` and `a[1]` respectively).

For convenience, the routine `numload` only returns the numeric array and `gisload` only loads the GIS data.

A.1.4 Online Help

When typing function names in a Python or *InfoPACK* shell, the Python interpreter will display the function syntax along with a full list of possible parameters. When working with Python Scripts, help concerning a particular routine can be quickly found by entering

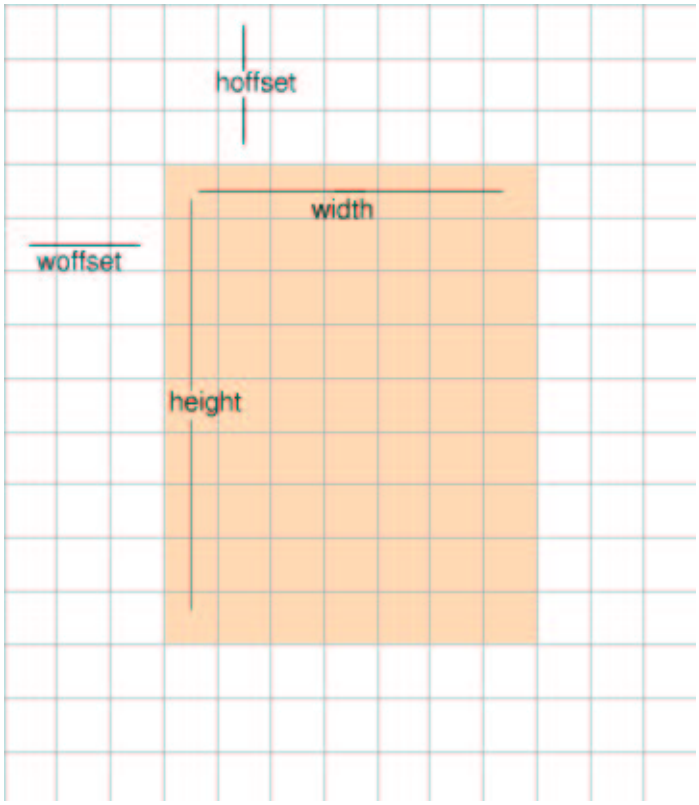

Figure A.1. *InfoPACK* naming conventions for image dimensions.

```
pydoc infopack.routine_name1
```

on the command line.

A.2 Common Parameters

Some parameters are common to all *InfoPACK* routines and are listed here.

A.2.1 Image dimensions

Figure A.1 shows the *InfoPACK* naming conventions as regards `width`, `height`, `woffset` and `hoffset`.

A.2.2 The Intensity Flag

Datasets for SAR images can be stored as either intensity (modulus or amplitude squared) or amplitude (square-root intensity) data. It is important that you know what type of data you are using. Many images are stored as amplitude since the speckle is less obtrusive when these images are displayed. *InfoPACK* routines assume by default that images are stored as amplitude. If they are not then the flag `intensity=1` needs to be set in the function or script call in order to indicate the image is in fact intensity data.

Internally, *InfoPACK* routines work on intensity data. If a routine is given amplitude data it will typically convert the data to intensity data, process the data and then convert the data back to amplitude. It is always worth checking the individual function specifications for exact behaviour though. For example, `colseg` run on amplitude data will return the square root of regional intensity mean values but will not square root the regional intensity gamma order parameters since that would make them meaningless.

A.2.3 Memory Management Options

All command line *InfoPACK* scripts (with very few exceptions) perform file to file operations by splitting the "infile" into tiles of size `tilesize`. These tiles are processed and then recombined into "outfile". An overlap can be used to remove the effect of any joins. Note that the default tile sizes and overlaps may need altering to better reflect the sizes of the features of interest in the processing chain. For example, when aiming to produce large regions with `texseg` or when running `segment` on polarimetric images, tile and overlap sizes will typically need to be increased to avoid any tiling artifacts appearing in the final image. By processing files in this manner, arbitrarily large images can be processed entirely in RAM.
Due to the file to file action, it is not possible to create "outfile" with the same name as "infile".
By default, intermediate results are written to a temporary directory as defined by system environment variables. Typically, this could be "`C:\WINNT\TEMP`" for a Windows system or "`/tmp/`" under Linux. Other directories can be used if preferred by changing the `tmpdir` parameter. The following parameters can be set to manage this process:

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Processing tiles with an overlap incurs an overhead in terms of disk space and file copying so routines that would not benefit from it (for instance `add.py` or `log10.py`) have the `overlap` set to a default of 0.
The tilesize and overlap defaults can be set within the GUI using the Configuration/Stripmap Options menu.

A.3 Documentation Conventions

In this chapter we document all of the high level user–callable facilities provided in *InfoPACK*. For each facility, both function (available from Python or from the *InfoPACK* console by importing module `infopack`) and script (available from a command line) versions are listed.

Image Parameters:

- For the script versions, the image files are given with the suffix `.nc`. These parameters must be the (full path) name of an image file.
- For the function version, the data files are given with the suffix `_array`. These parameters must be the name of Numeric Arrays containing the image data.

- For the script version, the specified file is written before exit. For the function version, the specified image is returned as the (or a) result of the function.

A.4 High Level Routines

A.4.1 segment

Purpose

`segment` takes an image and produces a 'cartoon' segmentation consisting of a collection of homogeneous regions. The regions are each coloured with its mean value (in the case of real or single layered complex images) or with the determinant of the M by M covariance matrix. For $M > 1$ complex images, this is defined as

$$det = \det |\Sigma|$$

where

$$\Sigma_{ij} = cov(x_i, x_j) \qquad i = 1, 2, \ldots M$$

is the covariance of x_i and x_j. The image or images need not emanate from a SAR processor: images from other sources (eg: optical) may be processed either alone or in combination with SAR images.
The algorithm starts with an adaptive rectangular (default: single pixel) segmentation of the image. The likelihood of the data fitting this segmentation can then be calculated. The judgement as to whether to move edge pixels depends on the change in likelihood of the segmentation. The process, governed by a simulated annealing optimisation algorithm, will eventually find the global optimal solution for the segmentation of the image. The computational cost increases with the area N of the image as $O(N \log N)$.

Command Line Script

segment.py `image.nc output.nc [quality=] [shape=] [stopping=] [cooling=] [looks=] [maxiters=] [regionsize=] [intensity=] [immutable=] [size=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = segment(image_array [,quality=] [,shape=] [,stopping=]
                       [,cooling=] [,looks=] [,maxiters=] [,regionsize=]
                       [,intensity=] [,immutable] [,size=] [,mtpol=])
```

Parameters

image.nc This can be either a single or a multiple layered real or complex-valued image. `segment` internally converts single layered complex images into single layered intensity images and processes them as with real images. A polarimetric image should be given as a layered complex-valued image.

output.nc The coloured region map output image is of the same dimensionality as the input image if the original image was real-valued. The coloured region map output image is single layered in the case of a complex-valued original image.

quality Chooses from a predefined set of segmentation parameters: quality = `default`, `sensitive` or `fast` (default: `default`) These choices set the following parameter values:

Parameter	Fast	Default	Sensitive
Shape	0.000001	0.000001	0.000001
Stopping	0.03	0.01	0.003
Cooling	30	10	1
Regionsize	1	1	1
Looks	-1	-1	-1
Maxiters	30	90	1000
Intensity	0	0	0

Any of these values can be over-ridden by subsequent values on the command line. The values `fast, default, sensitive` are provided as generally convenient settings, but because of the iterative nature of the segmentation process, and its data dependence, some initial parameter searching may be worthwhile for best results.

shape *float*, default 0.000001. The objective function minimised by `segment` includes a shape term designed to inhibit crinkle in the edges of regions. This works on a logarithmic scale so that smaller (eg 0.1, 0.01, 0.001) values produce smoother boundaries.

stopping *float*, default 0.01. Unless `maxiters` iterations are reached, the routine will terminate when two successive iterations show a relative change in the objective function of less than `stopping`.

cooling *float*, default 10. Determines the rate of cooling in the simulated annealing minimisation process. A larger value of `cooling` tends to decrease the time taken by the routine, but may increase the likelihood that it gets stuck in a local minimum.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

maxiters *integer*, default 90. The maximum number of iterations (complete scans of the image data) allowed.

regionsize *integer*, default 1. `segment` will start forming regions of size `regionsize` pixels. For most purposes the default value of 1 is recommended. A larger value will decrease runtimes but may lead to “chunky” segmentations.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

immutable This is an image (region map) whose edges will be preserved in the final segmentation. The image should be a single layer and have the same width and height as the input image. For the command line script, the `immutable=` option will be a file name and for the python procedure it is a Numeric array.

size *integer*, default 2. For polarimetric data only, this option sets the minimum region size.

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter

is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The `merge` routine may be employed to tidy the segmentation and ensure that all the edges are statistically significant.

A.4.2 `easyseg`

Purpose

Given an original image file, `easyseg` will produce a segmented image. Easyseg is designed as a simple interface to segmentation. It combines an optimal set of parameters to both `segment` and `merge`.

Command Line Script

easyseg.py `image.nc output.nc [looks=] [intensity=] [quality=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = easyseg(image_array [,looks=] [,intensity=] [,quality=]
                       [,mtpol=])
```

Parameters

image.nc This can be either a single or a multiple layered real or complex-valued image (depending on the chosen method). A polarimetric image should be given as a layered complex-valued image.

output.nc The coloured region map output image is of the same dimensionality as the input image if the original image was real-valued. The coloured region map output image is single layered in the case of a complex-valued original image.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

quality Chooses from a predefined set of segmentation parameters: quality = `default` (a good compromise), `sensitive` (preserve detail) or `fast` (quick result). The default is `default`.

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

A.4.3 texseg

Purpose

Segments the given input image according to the Blacknell-Tough measure.

$$\overline{I\log(I)}/\overline{I} - \log(\overline{I})$$

where I denotes the intensity image values. This measure accounts for variation of texture within the image. Joint segmentation is performed when there is more than one image layer.

Command Line Script

texseg.py `image.nc output.nc [quality=] [shape=] [stopping=] [cooling=] [looks=] [maxiters=] [regionsize=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = texseg(image_array [,quality=] [,shape=] [,stopping=]
                      [,cooling=] [,looks=] [,maxiters=]
                      [,regionsize=] [,intensity=])
```

Parameters

image.nc The file to be segmented. This can be either a single or a multiple layered real or complex-valued image. In this routine, all complex-valued layers of the image are converted to real valued layers before processing.

output.nc The output file containing the segmented image with regions coloured according to the Blacknell-Tough measure. The `texseg` region map output image is the same size as the input image.

quality Chooses from a predefined set of segmentation parameters: quality = `default` (a good compromise), `sensitive` (preserve detail) or `fast` (quick result). The default is `default`. These choices set the following parameter values:

Parameter	Fast	Default	Sensitive
Shape	0.0001	0.0001	0.0001
Stopping	0.01	0.01	0.003
Cooling	30	10	1
Looks	-1	-1	-1
Maxiters	30	90	1000
Regionsize	200	200	200
Intensity	0	0	0

Any of these values can be over-ridden by subsequent values on the command line. The values `fast, default, sensitive` are provided as generally convenient settings, but because of

the iterative nature of the segmentation process, and its data dependence, some initial parameter searching may be worthwhile for best results.

shape *float*, default 0.0001. The objective function minimised by `texseg` includes a shape term designed to inhibit crinkle in the edges of regions. This works on a logarithmic scale so that smaller (eg 0.1, 0.01, 0.001) values produce smoother boundaries.

stopping *float*, default 0.01. Unless `maxiters` iterations are reached, the routine will terminate when two successive iterations show a relative change in the objective function of less than `stopping`.

cooling *float*, default 10. Determines the rate of cooling in the simulated annealing minimisation process. A larger value of `cooling` tends to decrease the time taken by the routine, but may increase the likelihood that it gets stuck in a local minimum.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

maxiters *integer*, default 90. The maximum number of iterations (complete scans of the image data) allowed.

regionsize *integer*, default 200. `texseg` will start forming regions of a size `regionsize` pixels. Because texture is not defined for a single pixel, `regionsize` should be set greater than 1; for an initial look at a new dataset the default value of 200 is recommended.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

immutable This is an image (region map) whose edges will be preserved in the final segmentation. The image should be a single layer and have the same width and height as the input image. For the command line script, the `immutable=` option will be a file name and for the python procedure it is a Numeric array.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The output image is coloured with the Blacknell-Tough measure. The routine `colseg` may be used for a different colouring.

A.4.4 merge

Purpose

Given an original image file and a region map of this image, merge will unify any adjacent regions which (to within the specified probability of false alarm) are not significantly different in intensity. Alternatively, the minimum absolute value between regions may be specified.

Command Line Script

merge.py region_map.nc image.nc output.nc [method=] [cost=] [looks=] [intensity=] [crinkle=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]

Python Procedure

```
output_array = merge(region_map_array, image_array [,method=] [,cost=]
                     [,looks=] [,intensity=] [,crinkle=], [,mtpol=])
```

Parameters

region_map.nc The file containing the region map image. This image should be one of either

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image image.nc.
- A real-valued image of exactly the same dimensionality as image.nc.

In the case of a multiple layered region-map, the algorithm takes the union of all the edges in the region maps.

image.nc This can be either a single or a multiple layered real or complex-valued image. A polarimetric image should be given as a layered complex-valued image.

output.nc The merged region map output image is of the same dimensionality as the input image if the original image was real-valued. The merged region map output image is single layered in the case of a complex-valued original image.

method default "pfa". This selects the algorithm for comparing neighbouring regions.

- pfa Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used in determining whether to merge regions. Two adjacent regions will be merged if their intensities are not significantly different at a confidence level of pfa.
- crinkle Same as **pfa** but also smoothes the regions boundaries.
- abs Merges regions whose intensity differs by less than the absolute value set by the cost= option.
- crinkabs As **abs**, but also smoothes the region boundaries.
- rel Merges regions whose intensity differs by less than the relative value set by the cost= option.
- crinkrel As **rel**, but also smoothes the region boundaries.

`user` As **crinkle**, but the power of the smoothing can be set by the `crinkle=` parameter.

cost *float*, default 5. The definition of cost depends upon the chosen *method*. See *method* for details.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this number from the data.

crinkle *float*, default 4.0. This is an ad hoc parameter controlling how smooth the region boundaries should be. Increasing values between about 1 and 4 increase the amount of smoothing.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

None.

A.4.5 colseg

Purpose

Given an original image file and a region map of this image, colseg will produce a segmented image with the regions coloured according to one of a number of measures.

Command Line Script

colseg.py region_map.nc image.nc output.nc [method=] [looks=] [intensity=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]

Python Procedure

```
output_array = colseg(region_map_array, image_array [,method=]
                      [,looks=] [,intensity=] [,mtpol=])
```

region_map.nc The file containing the region map image. This image should be one of either

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image image.nc.
- A real-valued image of exactly the same dimensionality as image.nc.

In the case of a multiple layered region-map, the algorithm takes the intersection of the given layers to be the region map on which to perform the classification.

image.nc This can be either a single or a multiple layered real or complex-valued image (depending on the chosen method). A polarimetric image should be given as a layered complex-valued image.

output.nc The coloured region map output image is of the same dimensionality as the input image if the original image was real-valued. The coloured region map output image is single layered in the case of a complex-valued original image.

method The measure to be used to colour the regions (default "mean")

This string should be one of:

mean Mean of values within the segment. This measure may be applied on either real or complex images. For real images, the mean value is given as the intensity mean or the square root of the intensity mean depending on whether the data given is intensity or amplitude respectively.

inverse Inverse (1/mean) of mean option.

bt Blacknell-Tough measure within each segment:

$$bt = \overline{I\log(I)}/\overline{I} - \log(\overline{I})$$

where I denotes the intensity values. This measure is used in the textured segmentation routine texseg. This measure can only be prescribed for real images.

`gorder` Gamma order parameter of segments The estimated order of the gamma distribution best fitting the data in the segment. This measure can only be prescribed for real images.

`korder` K order parameter of segments The estimated order of the K distribution best fitting the data in the segment. Note that the default overlap size for this method is set to zero. Use of a non-zero overlap is likely to lead to unpredicatble behaviour due to the poor estimates for this measure from small sample sizes. This measure can only be prescribed for real images.

`size` Size of segments. Each pixel will have a value of the number of pixels in the region. This measure can only be prescribed for real images.

`region` Region number of the segment. This measure can only be prescribed for real images. Also this measure is not supported under the tiling scheme and will thus not run in a memory efficient fashion with arbitrarily large images.

`var` The variance of the data in the segment. This measure can only be prescribed for real images.

`det` The determinant of the M by M covariance matrix. For $M > 1$ complex images, this is defined as

$$det = \det |\Sigma|$$

where

$$\Sigma_{ij} = cov(x_i, x_j) \qquad i = 1, 2, \ldots M$$

is the covariance of x_i and x_j. This measure can only be prescribed for an M-layer ($M > 1$) complex image.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

Except for the `det` measure, each of these measures is computed layer-by-layer. With the exception of `det` and `mean`, the data must be real.

A.4.6 defleck

Purpose

Given an original image file and a region map of this image, `defleck` will remove small segments.

Command Line Script

defleck.py `region_map.nc image.nc output.nc [tilesize=] [overlap=] [tmpdir=] [size=] [mtpol=]`

Python Procedure

```
output_array = defleck(region_map_array, image_array [,size=] [,mtpol=])
```

Parameters

region_map.nc The file containing the region map image. This image should be one of either:

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image `image.nc`.
- A real-valued image of exactly the same dimensionality as `image.nc`.

In the case of a multiple layered region-map, the algorithm takes the intersection of the given layers to be the region map on which to perform the classification.

image.nc This can be either a single or a multiple layered real or complex-valued image (depending on the chosen method). A polarimetric image should be given as a layered complex-valued image.

output.nc The deflecked output image is of the same dimensionality as the input image if the original image was real-valued. The deflecked output image is single layered in the case of a complex-valued original image.

size *integer*, default 3. Regions with this number of pixels or less will be removed from the segmentation.

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

This routine is run automatically after `segment` with polarimetric data if the `size` parameter is set.

A.4.7 `classify`

Purpose

Given an original image and a region map of this image, `classify` will assign each region to one of a number of classes. The output is a single layer image with each region labelled with its class number (1,2,3 etc). The supervised classification scheme is based upon the assumption that the data is gamma distributed around the *a priori* class means. If necessary, order parameters for each class can be specified otherwise it is assumed that each class has the same order parameter.

Command Line Script

classify.py `region_map.nc image.nc output.nc means= [orders=] [scaling=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = classify(region_map_array, image_array ,means= [,orders=]
                        [,scaling=] [,intensity=])
```

Parameters

region_map.nc The file containing the region map image. This image should be one of either

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image `image.nc`.
- A real-valued image of exactly the same dimensionality as `image.nc`.

In the case of a multiple layered region-map, the algorithm takes the intersection of the given layers to be the region map on which to perform the classification.

image.nc A file containing the original image. This can be either a single or a multiple layered image (real or complex).

output.nc The output file. The output is a real-valued single layer image. Those regions in class 1 will have value 1, regions in class 2 will have value 2 and so on.

means Image of a priori class means per layer. The class means should be a single layered real valued image of height equal to the number of layers. The number of classes is then assumed to be equal to the width of the given image. For complex data, the width of the image is again equal to the number of classes, but to store the covariance matrix the height is the number of layers squared.

orders Optional image of a priori class order parameters per layer. The class orders should be a single layered real valued image of height equal to the number of layers. The number of classes is then equal to the width of the class orders image.

scaling *integer*, default 0. Flag to specify whether there should be scaling (1) or not (0). The purpose of the scaling option is to allow the temporal sequence for a given segment to be scaled to match the reference pattern in temporal signature classification. The absolute value of RCS is therefore normalised out and classification depends on the relative changes within the sequence alone.

This option is valuable when members of the same class have differing absolute RCS due to factors such as ground slope.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The `means.nc` and `orders.nc` files can be generated by either:

- Uniformly distributed class means can be calculated from the `mask_classes.py` script.
- Entering the values in a text file and using the `ascii2nc.py` utility. The values could be selected by using `histogram.py` to study the distribution of the data.
- By running the `mask_stats.py` utility on a set of given `a priori` class masks.
- By running the `mask_stats.py` utility on a set of given class masks made by hand (see the `images` directory for an example of this).
- For polarimetric data there is a `mask_polstats.py` which estimates the class means from the polarimetric covariance.

In any case, the `means.nc` and `orders.nc` files can be edited further to provide iterative supervised classification.

See Section B.5 for more routines on dealing with classifications.

A.4.8 change_detection

Purpose

For each region in a given region map, `change_detection` looks for significant change over a series of original speckled images.
The routine detects large–scale changes between two or more images; *large–scale* implies that it is not tailored to detecting differences in point targets between images. The images to be scanned for changes must be provided as layers in a single multi–layer image (if your images are separate, use `layer_combine` to build them into a single 3-D image).
The routine uses the segmentation of each layer to carry out its analysis, and this must be provided too. In most cases the "joint segmentation" produced by segmenting the multilayer image is what will most conveniently be used.
The routine detects changes across the whole series of provided layers - the ordering of the layers in the multiple layer image file is not important. The output is a single layer image specifying where any significant changes were found over the whole sequence.
If you are interested in the individual changes found between adjacent pairs of layers in the input image, use routine `change_detection_position`.

Command Line Script

change_detection.py `region_map.nc image.nc output.nc [pfa=] [mlthresh=] [looks=] [intensity=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = change_detection(region_map_array, image_array [,pfa=]
                                [,mlthresh=] [,looks=] [,intensity=]
                                [,mtpol=])
```

Parameters

region_map.nc The file containing the region map image. This image should be one of either

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image `image.nc`.
- A real-valued image of exactly the same dimensionality as `image.nc`.

In the case of a multiple layered region-map, the algorithm takes the intersection of the given layers to be the region map on which to perform the change detection.

image.nc This image must be a multiple layered (i.e. more than 1 layer) real or complex valued image.

output.nc The output file. The output is real-valued single layer image, with dimensionality equal to the first layer of the original image. Pixels are either coloured with value 1 (indicating that they belong to regions for which there has been significant change) or with value 0 (that they don't).

pfa *float*, default 6. Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used

in determining whether changes are significant. The larger pfa is, the less chance there will be of reporting insignificant changes.

pfa *float*, default 0. If this is non-zero, threshold the likelihood directly rather than calculating the probability. The `pfa` option will be ignored.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

If you are interested in the individual changes found between adjacent pairs of layers in the input image, use routine `change_detection_position`.

A.4.9 change_detection_position

Purpose

For each region in a given region map, change_detection_position looks for significant change over a series of original speckled images, but in more detail than does change_detection. The routine detects large–scale changes between adjacent pairs of images; *large–scale* implies that it is not tailored to detecting differences in point targets between images. The images to be scanned for changes must be provided as layers in a single multi–layer image (if your images are separate, use layer_combine to merge them into a single 3-D image).
The routine uses the segmentation of each layer to carry out its analysis, and this must be provided too. In most cases the "joint segmentation" produced by segmenting the multilayer image is what will most conveniently be used.
The routine detects changes between adjacent pairs of layers in the provided image - the ordering of the layers in the multiple layer image file is therefore crucial. The output is a multiple layer image with number of layers equal to the number of adjacent pairs in the sequence of layers in the original image (i.e. number of layers minus 1). Each of these layers indicates where significant changes were found between the image pair.
If you are interested solely in whether there are changes over the whole sequence of layers in the input image, use the routine change_detection.
This routine is useful for example when monitoring the growth of crops over the seasons.

Command Line Script

change_detection_position.py region_map.nc image.nc output.nc [pfa=] [looks=] [intensity=] [mtpol=] [tilesize=] [overlap=] [tmpdir=]

Python Procedure

```
output_array = change_detection_position(region_map_array, image_array
                                         [,pfa=] [,mlthresh=] [,looks=]
                                         [,intensity=] [,mtpol=])
```

Parameters

region_map.nc The file containing the region map image. This image should be one of either

- A single layered real-valued image of dimensions equal to the dimensions of the first layer of the original image image.nc.
- A real-valued image of exactly the same dimensionality as image.nc.

In the case of a multiple layered region-map, the algorithm takes the intersection of the given layers to be the region map on which to perform the change detection.

image.nc This image must be a multiple layered (i.e. more than 1 layer) real or complex valued image.

output.nc The output file. The output is real-valued layer image, with dimensionality equal to

that of the original image minus one of its layers. Thus each output layer corresponds to changes between each adjacent pair of layers in the original image. Pixels are either coloured with value 1 (indicating that they belong to regions for which there has been significant change) or with value 0 (that they don't).

pfa *float*, default 6. Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used in determining whether changes are significant. The larger pfa is, the less chance there will be of reporting insignificant changes.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

If you are interested solely in whether there are changes over the whole sequence of layers in the input image, use the routine `change_detection`.

A.4.10 `despeckle`

Purpose

`despeckle` takes a speckled image and estimates the underlying radar cross-section using a variant of the simulated annealing optimisation method. The resulting despeckled image is smoothly varying yet preserves detail.

Command Line Script

despeckle.py `image.nc output.nc [quality=] [cooling=] [looks=] [maxiters=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = despeckle(image_array [,quality=] [,cooling=] [,looks=]
                         [,maxiters=] [,intensity=])
```

Parameters

image.nc The image to be smoothed. This can be either a single or a multiple layered real image.

output.nc The smoothed output image is of the same dimensionality as the input image, yet is always real-valued.

quality Chooses from a predefined set of segmentation parameters: quality=`default`, `sensitive` or `fast` (default: `default`). These choices set the following parameter values:

Parameter	Fast	Default	Sensitive
Cooling	0.75	0.75	1
Looks	-1	-1	-1
Maxiters	25	40	50
Intensity	0	0	0

Any of these values can be over-ridden by subsequent values on the command line. The values `fast`, `default`, `sensitive` are provided as generally convenient settings, but because of the iterative nature of the smoothing process, and its data dependence, some initial parameter searching may be worthwhile for best results.

cooling *float*, default 0.75. Determines the rate of cooling in the simulated annealing minimisation process. A larger value of `cooling` tends to decrease the time taken by the routine, but may increase the likelihood that it gets stuck in a local minimum.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

maxiters *integer*, default 40. The maximum number of iterations (complete scans of the image data) allowed.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

None.

A.4.11 `filter`

Purpose

Uses a moving window of size `width` by `height` to filter the image by computing an output pixel value from within each window according to the specified `method`.

Command Line Script

filter.py `image.nc output.nc [method=] [width=] [height=] [looks=] [pfa=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = filter(image_array [,method=] [,width=] [,height=]
                      [,looks=] [,pfa=] [,intensity=])
```

Parameters

image.nc This can be either a single or a multiple layered real or complex-valued image (depending on the chosen method). A polarimetric image should be given as a layered complex-valued image.

output.nc The coloured region map output image is of the same dimensionality as the input image if the original image was real-valued. The coloured region map output image is single layered in the case of a complex-valued original image.

method This string specifies the filter function to be used (default `mean`).

The following methods are available:

`mean` This measure may be applied on either real or complex images. For real images, the mean value is given as the intensity mean or the square root of the intensity mean depending on whether the data given is intensity or amplitude respectively.

`bt` Blacknell-Tough measure within each segment:

$$bt = \overline{I\log(I)}/\overline{I} - \log(\overline{I})$$

where I denotes the intensity values. This measure can only be prescribed for real images. This measure is used in the textured segmentation routine `texseg`.

`gorder` Gamma order parameter of segments The estimated order of the gamma distribution best fitting the data in the window. This measure can only be prescribed for real images.

`korder` K order parameter of segments. The estimated order of the K distribution best fitting the data in the window. This measure can only be prescribed for real images. The parameter `looks` can be used with this method.

`enl` Equivalent number of looks (ENL) calculated from the squared mean divided by the variance. This measure can only be prescribed for real images.

`var` The variance of the data in the window. This measure can only be prescribed for real images.

`med` The median of the data in the window. This measure can only be prescribed for real images. The filter determines the median for the data given, i.e. it ignores the intensity flag.

`det` The determinant of the M by M covariance matrix. For $M > 1$ complex images, this is defined as

$$det = \det |\Sigma|$$

where

$$\Sigma_{ij} = cov(x_i, x_j) \qquad i = 1, 2, \ldots M$$

is the covariance of x_i and x_j. This measure can only be prescribed for an M-layer ($M > 1$) complex image.

`cfar` Single pixel CFAR target detector. The CFAR algorithm works by calculating the likelihood of the central pixel in the window coming from the background distribution (as represented by the rest of the window). The classification of the central pixel as a target depends upon the comparison of this likelihood with the threshold calculated from the probability of false alarm (`pfa`). A lower value of probability of false alarm will mean less targets will be detected. Since the result is determined by the accuracy of the statistical estimates, it may be advisable to have a larger data sample, i.e. window size, than the default. The parameters `pfa` and `looks` can be used with this method.

`scan` Scanning edge detector. This edge detector works by calculating the significance of every possible edge position in every window. Since the result is determined by the accuracy of the statistical estimates, it may be advisable to have a larger data sample, i.e. window size, than the default. The parameters `pfa` and `looks` can be used with this method. A lower value of probability of false alarm (`pfa`) will mean less edges will be detected.

`mmse` The Minimum mean square error (MMSE) despeckling filter. This measure can only be prescribed for real images and uses the `looks` parameter.

`lee` The J.S. Lee despeckling filter. This measure can only be prescribed for real images and uses the `looks` parameter.

width *integer*, default 3. The width of the filter window.

height *integer*, default 3. The height of the filter window.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this number from the data. This estimate can be used with some of the methods.

pfa *float*, default 5. Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used in determining whether changes are significant. The larger pfa is, the less chance there will be of reporting insignificant changes. This estimate can be used with some of the methods.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the

same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

Conventient wrappers to this function are available in the routines `filter_bt`, `filter_det`, `filter_enl`, `filter_gorder`, `filter_korder`, `filter_mean`, `filter_var`, `median` and `scanedge`.

A.4.12 `filter_bt`, `filter_det`, `filter_enl`, `filter_gorder`, `filter_korder`, `filter_mean`, `filter_var`

Purpose

Uses a moving window of size `width` by `height` to filter the image by computing an output pixel value from within each window according to the `method` indicated in the routine name.

Command Line Script

filter_method.py `image.nc output.nc [width=] [height=] [looks=] [pfa=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

`Use filter with the method= parameter defined appropriately.`

Parameters

See parameter descriptions for `filter` in section A.4.11.

A.4.13 `cfar`

Purpose

Single pixel CFAR target detector. The CFAR algorithm works by calculating the likelihood of the central pixel in a moving window (of size `width` by `height`) coming from the background distribution (as represented by the rest of the window). The classification of the central pixel as a target depends upon the comparison of this likelihood with the threshold calculated from the probability of false alarm (`pfa`). A lower value of probability of false alarm will mean less targets will be detected.

Since the result is determined by the accuracy of the statistical estimates, it may be advisable to have a larger data sample, i.e. window size, than the default.

Command Line Script

cfar.py `image.nc output.nc [width=] [height=] [looks=] [pfa=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = filter(image_array, method="cfar" [,width=] [,height=]
                      [,looks=] [,pfa=] [,intensity=])
```

Parameters

image.nc This can be either a single or a multiple layered real valued image.

output.nc The coloured region map output image is of the same dimensionality as the input image if the original image was real-valued. The coloured region map output image is single layered in the case of a complex-valued original image.

width *integer*, default 3. The width of the filter window.

height *integer*, default 3. The height of the filter window.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

pfa *float*, default 5. Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used in determining whether changes are significant. The larger pfa is, the less chance there will be of reporting insignificant changes.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

None.

Appendix B

Low Level and Utility Routine Documentation

B.1 Documentation Conventions

In this chapter we document all of the low level user–callable facilities provided in *InfoPACK*. For each facility, both function (available from Python or from the *InfoPACK* console by importing module `infopack`) and script (available from a command line) versions are listed; for a discussion of the differences see sections 1.1 and A.1. Also see section A.2 for a description of the common parameters.

Image Parameters:

- For the script versions, the image files are denoted `image`. These parameters must be the (full path) name of an image file.
- For the function version, the data files are denoted `array`. These parameters must be the name of Numeric Arrays containing the image data.
- Input parameters are given suffix `_in`
- Output parameters are given suffix `_out`. For the script version, the specified file is written before exit. For the function version, the specified image is returned as the (or a) result of the function.

B.1.1 `pydoc`

Purpose

The Python documentation tool.

Command Line Script

```
pydoc routine
```

Python Procedure

```
pydoc.help(routine)
```

Parameters

routine Python comes with extensive on-line help. `pydoc infopack` gives an overview of the Python scripts available in InfoPACK, while for instance `pydoc infopack.segment` gives details of the particular routine.

B.2 Image Manipulation Routines

B.2.1 add

Purpose

Add two image files. The result is the sum of corresponding pixel values of the input images.

Command Line Script

```
add.py image1_in image2_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.add(array1_in, array2_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.2 between

Purpose

Image pixels that lie between the upper and lower limits are set to a specified true value, otherwise they are set to a false value. Upper and lower limits must be given.

Command Line Script

```
between.py image_in image_out upper= lower= [true=] [false=] [tilesize=]
[overlap=] [tmpdir=]
```

Python Procedure

```
array_out = between(array_in, upper=, lower= [,true=] [,false=])
```

Parameters

true *float*, default 1.

false *float*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.3 `clip`

Purpose

Clips the pixels in an image with given minimum (0) and maximum (1) values.

Command Line Script

```
clip.py image_in image_out [min=][, max=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = clip(array_in, [min=][, max=])
```

Parameters

minimum minimum value 0

maximum maximum value 1

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.4 `divide`

Purpose

Divides two image files. The result is the division of corresponding pixel values of the input images.

Command Line Script

```
divide.py image1_in image2_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.divide(array1_in, array2_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.5 `edge`

Purpose

Given a region map, `edge` will scan the segments and construct an edge map

Command Line Script

edge.py `region_map.nc output.nc [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = edge(region_map_array)
```

Parameters

region_map.nc The file containing the region map image. This should be a single or multiple layered real-valued image.

output.nc The output file. This is a real-valued image of the same dimensionality as the input region map.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

`edge` is only applicable to segmented images. Using `edge` to produce an edge map of other than a segmented image will produce disapointing results. Instead the routine `scanedge` should be used.

B.2.6 fill

Purpose

Creates a real valued image of the given dimensions (width, height and the number of layers) and colours all pixels with the same uniform value. Width and height parameters must be specified.

Command Line Script

```
fill.py image_out width= height= [layers=] [value=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = fill(width=, height= [,layers=] [,value=])
```

Parameters

layers *integer*, default 1.

value *float*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.7 fill_gammarand

Purpose

Creates a real valued image of the given dimensions (width, height and the number of layers) filled with random samples. These samples have a gamma distribution with the given mean and order parameter. Seeds ”seed1” and ”seed2” may be specified to give a reproducible sequence of random numbers. Width and height parameters must be specified.

Command Line Script

```
fill_gammarand.py image_out width= height= [layers=] [mean=] [order=]
[seed1=] [seed2=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = fill_gammarand(width=, height= [,layers=] [,mean=] [,order=]
[,seed1=] [,seed2=])
```

Parameters

layers *integer*, default 1.

mean *float*, default 1.

order *float*, default 1.

seed1 *integer*, default 0.

seed2 *integer*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.8 `fill_gaussrand`

Purpose

Creates a real valued image of given dimensions (width, height and the number of layers) filled with random samples. These samples have a gaussian distribution with the given mean and standard deviation. Seeds ”seed1” and ”seed2” may be specified to give a reproducible sequence of random numbers. Width and height parameters must be specified.

Command Line Script

```
fill_gaussrand.py image_out width= height= [layers=] [mean=] [sd=]
[seed1=] [seed2=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = fill_gaussrand(width=, height= [,layers=] [,mean=] [,sd=]
[,seed1=] [,seed2=])
```

Parameters

layers *integer*, default 1.

mean *float*, default 0.

sd *float*, default 1.

seed1 *integer*, default 0.

seed2 *integer*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.9 `fill_unirand`

Purpose

Creates a real valued image of given dimensions (width, height and the number of layers) filled with random samples. These samples have a uniform distribution between 0 and 1. Seeds ”seed1” and ”seed2” may be specified to give a reproducible sequence of random numbers. Width and height parameters must be specified.

Command Line Script

```
fill_unirand.py image_out width= height= [layers=] [seed1=] [seed2=]
[tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = fill_unirand(width=, height= [,layers=] [,seed1=] [,seed2=])
```

Parameters

layers *integer*, default 1.

seed1 *integer*, default 0.

seed2 *integer*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.10 `fill_polrand`

Purpose

Creates a complex valued image of given dimensions (width, height) filled with random samples. These samples are derived from the Cholesky decomposition of the `covar` input and so a 3×3 input matrix will produce a 3 layered complex image. Seeds ”seed1” and ”seed2” may be specified

to give a reproducible sequence of random numbers. The width, height and covar parameters must be specified.

Command Line Script

```
fill_unirand.py image_out width= height= [covar=] [seed1=] [seed2=]
[tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = fill_polrand(width=, height=, covar= [,seed1=] [,seed2=])
```

Parameters

covar The covariance matrix can either be specified on the command line. For instance `covar=1,0,0,0,1+3j,0,0,0,0.1+2j` (Note no spaces around commas) or the covariance matrix can be in a file as with `covar=covmat.nc`. For a 3×3 covariance matrix, the output image with have 3 complex layers.

seed1 *integer*, default 0.

seed2 *integer*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.11 georef

Purpose

Returns a geocoded image by shifting pixels according to x and y deltas. These deltas must be stored as short (16 bit) and the output image will be the same size as the two delta images.

Command Line Script

```
polyscan.py image_in image_out [xaverage=] [yaverage=] [intensity=] [blank=]
[xoffset=] [yoffset=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = georef(array_in [,xaverage=] [,yaverage=] [,intensity=]
                   [,blank=] [,xoffset=] [,yoffset=])
```

Parameters

xaverage *integer*, default 1. Averages this many pixels horizontally.

yaverage *integer*, default 1. Averages this many pixels vertically.

blank *float*, default 0. Value to use for the background.

xoffset *integer*, default 0. Add this value to the x deltas.

yoffset *integer*, default 0. Add this value to the y deltas.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.12 `layer_sum`

Purpose

Adds all the layers in two or more images. The result is the sum of corresponding pixel values in all of the layers of the input images.

Command Line Script

```
layer_sum.py image1_in image2_in ...  imagen_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = layer_sum((array1_in, array2_in, ..., arrayn_in))
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The width and height of the input images must be the same, although they can differ in the number of layers. The output image will have the same width and height and only be a single layer image.

B.2.13 log10

Purpose

The result is the logarithm to the base 10 of each individual pixel value from the input image.

Command Line Script

```
log10.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.log10(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.14 loge

Purpose

The result is the natural logarithm of each individual pixel value from the input image.

Command Line Script

```
loge.py image1_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.loge(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.15 median

Purpose

Median filter. For each pixel, the median value in the surrounding window is returned.

Command Line Script

```
median.py image_in image_out [width=] [height=] [tilesize=]
[overlap=] [tmpdir=]
```

Python Procedure

```
array_out = filter(array_in, method='med' [,width=] [,height=])
```

Parameters

width *integer*, default 3. The width of the filter window.

height *integer*, default 3. The height of the filter window.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.16 multiply

Purpose

Multiplies two image files. The result is the division of the corresponding pixel values between the two input images.

Command Line Script

```
multiply.py image1_in image2_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.multiply(array1_in, array2_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.17 pca

Purpose

Perform Principal Component Analysis on an image.

Command Line Script

```
pca.py image_in image_out [,white=] [,posdef=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = pca(array_in [,white=] [,posdef=])
```

Parameters

white *integer*, Default 0. If `white` is non-zero then also perform whitening (Divide the output by the square-root of the eigen values.

posdef *integer*, Default 0. If `posdef` is non-zero then assume the covariance matrix is a square, Hermitian positive definite matrix.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.18 pol2covar

Purpose

Converts polarimetric data from (HH, HV, VV, etc) to covariance format HHHH*, HHHV*, HHVV*, ...

Command Line Script

```
pol2covar.py image_in image_out [mtpol=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = pol2covar(array_in, [,mtpol=])
```

Parameters

mtpol *integer*, default 0. Polarimetric data is stored as layers of complex data, however this is the same file format as multi-temporal polarimetric data. To distinguish the two, the `mtpol` parameter is used. For instance, 5 days of 3 channel polarimetric data would be stored as a 15 layer complex image and the `mtpol=3` parameter would be used. The default value of `mtpol=0` uses all of the layers as a single polarimetric image.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.19 `polyscan`

Purpose

Draws a closed polygon on an image.

Command Line Script

```
polyscan.py image_in image_out polygon= [value=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = polyscan(array_in, polygon= [,value=]
```

Parameters

polygon *list*. This is a list of (x,y) pixel coordinates of the vertices. For example `polygon=10,10,100,10,10,100` (Note no spaces are allowed around the commas).

polygon *float*. Default 0. The enclosed area of the polygon is filled with this value.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.20 `resize`

Purpose

Resize an existing image. The resulting image is resized to the new width and height, using a

specified method of interpolation.

Command Line Script

```
resize.py image_in image_out [width=] [height=] [wscale=] [hscale=]
          [method=]
```

Python Procedure

```
array_out = resize(array_in [,width=] [,height=] [,wscale=] [,hscale=]
                   [,method=])
```

Parameters

width *integer*, default None. The new image width.

height *integer*, default None. The new image height.

wscale *integer*, default 1.
The width parameter can be also be specified as a scaling of the original image size with the parameters wscale. For example, 'wscale=0.5' would produce an image half the width of the original image.

hscale *integer*, default 1. The height parameter can be also be specified as a scaling of the original image size with the parameters hscale.

method Choose from the following methods of interpolation:
"`resample`" : Nearest neighbour.
"`average`": Averages neighbouring pixels when shrink, nearest neighbour when expanding image.
"`linear`": Linear.
"`fft`" : Fourier transform (sinc).
The default method is 'resample'.

B.2.21 `scalar_add`

Purpose

The result is the individual pixel values from the input image plus a specified constant.

Command Line Script

```
scalar_add.py image_in image_out scalar= [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = scalar_add(array_in, scalar=)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.22 scalar_divide

Purpose

The result is the individual pixel values from the input image divided by a specified constant.

Command Line Script

```
scalar_divide.py image_in image_out scalar= [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = scalar_divide(array_in, scalar=)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.23 scalar_multiply

Purpose

The result is the individual pixel values from the input image multiplied by a specified constant.

Command Line Script

```
scalar_multiply.py image_in image_out scalar= [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = scalar_multiply(array_in, scalar=)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.24 scanedge

Purpose

Scanning edge detector. This edge detector works by calculating the significance of every possible edge position within a moving window of size `width` by `height`.

Since the result is determined by the accuracy of the statistical estimates, it may be advisable to have a larger data sample, i.e. window size, than the default. A lower value of probability of false alarm (`pfa`) will mean less edges will be detected.

Command Line Script

scanedge.py `image.nc output.nc [width=] [height=] [looks=] [pfa=] [intensity=] [tilesize=] [overlap=] [tmpdir=]`

Python Procedure

```
output_array = filter(image_array, method="scan" [,width=] [,height=]
                      [,looks=] [,pfa=] [,intensity=])
```

Parameters

image.nc This should be either a single or a multiple layered real image.

output.nc This is a 0/1 image with the edges marked with 1.

width *integer*, default 3. The width of the filter window.

height *integer*, default 3. The height of the filter window.

looks *float*, default -1. The looks measure of the image data. If `looks` is negative (use -1) the routine will compute this parameter from the data.

pfa *float*, default 5. Specifies the Probability of False Alarm ($= 10^{-pfa}$ if $pfa > 1$) to be used in determining whether changes are significant. The larger pfa is, the less chance there will be of reporting insignificant changes.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

This routine is actually a wrapper to the `filter` function of Section A.4.11.

B.2.25 `sqrt`

Purpose

The result is the square root of individual pixel values from the input image.

Command Line Script

```
sqrt.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = sqrt(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.26 `square`

Purpose

The result is the square of individual pixel values from the input image.

Command Line Script

```
square.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = square(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.27 subtract

Purpose

Subtracts two image files. The result is the subtraction of corresponding pixel values of the input images.

Command Line Script

```
subtract.py image1_in image2_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = Numeric.subtract(array1_in, array2_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.2.28 sum

Purpose

Adds two or more images. The result is the sum of corresponding pixel values of the input images.

Command Line Script

```
sum.py image1_in image2_in ...  imagen_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = listsum((array_in, image2_in, ..., arrayn_in))
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The dimensions (layers, width, height) of the input images must be the same. The output image will have the same dimensions.

B.2.29 `threshold`

Purpose

Image pixels that lie above the threshold value are set to a specified true value, otherwise they are set to a false value. The threshold value must be given.

Command Line Script

```
threshold.py image_in image_out threshold= [true=] [false=] [tilesize=]
             [tmpdir=]
```

Python Procedure

```
array_out = threshold(array_in, threshold= [,true=] [,false=])
```

Parameters

threshold *float*, must be specified.

true *float*, default 1.

false *float*, default 0.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.3 File Handling Facilities

B.3.1 `ascii2nc`

Purpose

Converts an ASCII file into an *NASoftware Ltd NetCDF* file.
This can be useful for generating means and orders files for use in supervised classification. Each line in the ASCII file corresponds to one row of the array. Numbers within a line are separated by white space.

Command Line Script

```
ascii2nc.py ascii_in image_out
```

Python Procedure

```
None provided.
```

B.3.2 `convert`

Purpose

A utility to convert files between different data types and various GDAL (Geospatial Data Abstraction Library) formats. The Python procedure provides data type translation, the command line script also allows for specifying a GDAL output format.

Command Line Script

```
convert.py image_in image_out [method=] [format=]
```

Python Procedure

```
array_out = convert(array_in [,method=])
```

Parameters

method *string*, default `None`.
`None`: Copy the data (useful for filetype conversion).
`byte`: Convert image to byte data (Numeric Int0).
`short`: Convert image to short data (Numeric Int16).
`int`: Convert image to int data (Numeric Int32).
`float`: Convert image to float data (Numeric Float32).
`double`: Convert image to double data (Numeric Float).

`complex`: Convert image to float complex data (Numeric Complex32).
`dcomplex`: Convert image to double complex data (Numeric Complex).
`real`: Real part of complex data.
`imag`: Imaginary part of complex data.
`conj`: Conjugate of complex data.
`abs`: Amplitude (modulus) of complex data.
`power`: Power (modulus squared) of complex data.
`phase`: Phase of complex data.

format The following formats are supported:
`VRT`: Virtual Raster
`GTiff`: GeoTIFF
`HFA`: Erdas Imagine Images (.img)
NASoftware Ltd: *NASoftware Ltd* NetCDF Images (.nc)
`SAR_CEOS`: CEOS SAR Image
`CEOS`: CEOS Image
`ELAS`: ELAS
`AIG`: Arc/Info Binary Grid
`AAIGrid`: Arc/Info ASCII Grid
`SDTS`: SDTS Raster
`DTED`: DTED Elevation Raster
`PNG`: Portable Network Graphics
`JPEG`: JPEG JFIF
`MEM`: In Memory Raster
`JDEM`: Japanese DEM (.mem)
`GIF`: Graphics Interchange Format (.gif)
`ESAT`: Envisat Image Format (.N1)
`BSB`: Maptech BSB Nautical Charts
`XPM`: X11 PixMap Format (.xpm)
`PNM`: Portable Pixmap Format (netpbm)
`DOQ1`: USGS DOQ (Old Style)
`DOQ2`: USGS DOQ (New Style)
`ENVI`: ENVI .hdr Labelled
`EHdr`: ESRI .hdr Labelled
`PAux`: PCI .aux Labelled
`MFF`: Atlantis MFF Raster
`MFF2`: Atlantis MFF2 (HKV) Raster
`EFF`: Eosat Fast Format
`FujiBAS`: Fuji BAS Scanner Image
`FIT`: FIT image
`USGSDEM`: USGS Optional ASCII DEM
`GXF`: GeoSoft Grid Exchange Format

The default output format is *NASoftware Ltd.*

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The command

```
convert.py image.nc output.tif format='GTiff'
```

may not produce a displayable TIFF image. The GeoTiff format allows for floating point data to be stored in the file which many image display programs cannot interpret.

B.3.3 extract

Purpose

Returns a sub-image of given dimensions at the given offsets. The width and height must be specified.

Command Line Script

```
extract.py image_in image_out width= height= [woffset=] [hoffset=]
```

Python Procedure

```
array_out = extract(array_in, width=, height= [,woffset=] [,hoffset=])
```

Parameters

woffset *integer*, default 0.

hoffset *integer*, default 0.

B.3.4 gdal_translate

Purpose

A utility to convert files between various GDAL (Geospatial Data Abstraction Library) formats. The default output format is GeoTIFF. There is limited support for resizing images, selecting subwindows, and converting between data types. Typing `gdal_translate` provides a full help message.

Command Line Script

```
gdal_translate [-of format] [-ot type] image_in image_out
```

Python Procedure

```
None provided.
```

B.3.5 gisload

Purpose

Returns the class containing the GIS information stored in an *NASoftware Ltd NetCDF* file.

Command Line Script

```
None provided.
```

Python Procedure

```
gis_class = gisload('image_in')
```

Further Comments

InfoPACK provides little support for manipulating the GIS data, instead it is usually just passed to the `save` routine.

B.3.6 gissave

Purpose

Saves/updates the class containing the GIS information stored in an *NASoftware Ltd NetCDF* file.

Command Line Script

```
None provided.
```

Python Procedure

```
gissave('image_out', gis_class)
```

B.3.7 insert

Purpose

Inserts a sub-image into a larger image at the given offsets.

Command Line Script

```
insert.py image_in insert_in image_out [woffset=] [hoffset=]
```

Python Procedure

See save B.3.17.

Parameters

woffset *integer*, default None.

hoffset *integer*, default None.

B.3.8 layer_combine

Purpose

Combines image layers (held in a set of seperate images) into a single file. All input images must be of the same width and height.

Command Line Script

```
layer_combine.py image1_in image2_in ...  image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out= layer_combine((array1_in, array2_in, ...  arrayn_in))
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.3.9 layer_split

Purpose

Splits a multi-layered image into a numbered set of images corresponding to the individual layers. Results are written into the files "image_out_000.nc", "image_out_001.nc", ...

Command Line Script

```
layer_split.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = layer_split(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The python procedure returns a list of arrays.

B.3.10 load

Purpose

Loads an image file and its GIS information (if available).

`Load` returns a list of length two. The first element is the GDAL file as a Numeric array, while the second element is a class containing the GIS information.

`Load` loads the subimage of dimensions width by height starting at the pixel location given by woffset and hoffset.

Command Line Script

```
None provided.
```

Python Procedure

```
array_out, gis_class = load('image_in' [,woffset=] [,hoffset=] [,width=]
[,height=])
```

Parameters

woffset *integer*, default 0.

hoffset *integer*, default 0.

width *integer*, default None = read whole image in.

height *integer*, default None = read whole image in.

B.3.11 `makebitmap`

Purpose

Converts a Numeric array to an RGB array. The array can then be saved in a variety of formats suitable for display in common graphics packages.

For each layer of the image, values that lies in the range `min` to `max` are mapped into the range 0 to 255. If `min` or `max` are not specified then the parameter `sd` is used with the following mapping:

$\mu - \texttt{sd} \times \sigma \rightarrow 0$
$\mu + \texttt{sd} \times \sigma \rightarrow 255$

where μ is the mean and σ is the variance of the data. A value of `sd` = 0 has the special meaning of

minimum $\rightarrow 0$
maximum $\rightarrow 255$

Complex data cannot be directly displayed as a bitmap and so must be converted to real data first. This is done by the option `cmpl2real`.

Command Line Script

```
See tojpeg.py, totiff.py.
```

Python Procedure

```
array_out = makebitmap(array_in [,cmpl2real=] [,sd=] [,min=] [,max=]
[,verbose=])
```

Parameters

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:

"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).

"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

sd *float*, default 2.0.

min *list*, default None. This can either be a single value or a list. eg `min=3` or `min=(0,3.7,-2)`.

max *list*, default None.

verbose *integer*, default 0. If set to 1 then the estimated `min` and `max` values are printed out.

B.3.12 `nanfix`

Purpose

Sets NaN or infinities to a specifed value.

Command Line Script

```
nanfix.py image_in image_out [nan=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = nanfix(array_in [,nan=])
```

Parameters

nan *float*, default 0. Wherever a NaN or infinity occurs in the image is replaced with this value.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.3.13 `nc2ascii`

Purpose

Converts an *NASoftware Ltd NetCDF* file into an ASCII file.

Command Line Script

```
nc2ascii.py image_in ascii_out
```

Python Procedure

```
None provided.
```

B.3.14 nc2raw

Purpose

Converts an *NASoftware Ltd NetCDF* file into a raw (binary) data file.

Command Line Script

```
nc2raw.py image_in raw_out [,byteswap=]
```

Python Procedure

```
None provided.
```

Parameters

byteswap *integer*, default 0. The option `byteswap=1` transforms big-endian to little-endian data.

B.3.15 numload

Purpose

Loads an image file, without the GIS information.

`numload` returns the GDAL file as a Numeric array. `numload` loads the subimage of dimensions `width` by `height` starting at the pixel location given by `woffset` and `hoffset`.

Command Line Script

```
None provided.
```

Python Procedure

```
array_out = numload('image_in' [,woffset=] [,hoffset=] [,width=] [,height=])
```

Parameters

woffset *integer*, default 0.

hoffset *integer*, default 0.

width *integer*, default None = read whole image in.

height *integer*, default None = read whole image in.

B.3.16 raw2nc

Purpose

Converts a binary data file into an *NASoftware Ltd NetCDF* file. The type of data must be specified.

Command Line Script

```
raw2nc.py raw_in image_out type= [byteswap=] [header=] [layers=] [verbose=]
```

Python Procedure

```
None provided
```

Parameters

type *string*, must be specified.
Type is one of:

- 'D' Complex, Complex64
- 'F' Complex0, Complex16, Complex32, Complex8
- 'd' Float, Float64
- 'f' Float0, Float16, Float32, Float8
- 'l' Int
- 'b' Int0, Int8
- 's' Int16
- 'i' Int32

byteswap *integer*, default 1. When set to 1 the routine transforms big-endian to little-endian data.

Depending on how your images are constructed, it is possible that you will need to reverse the order of the bytes. To see whether this is the case, it is suggested that you look at the statistics of the converted image to check that the values it reports look sensible; if not, you probably need to swap the byte order. A visual inspection may also prove sufficient.

header *integer*, default 0. When set to 0 the routine skips the header (bytes).

layers *integer*, default 1. To set the number of layers - Band sequential (BSQ) format.

verbose *integer*, default 0. When set to 0 the routine turns off information messages.

B.3.17 `save`

Purpose

Saves the Numeric array and class containing the GIS information in an *NASoftware Ltd NetCDF* file. Options are available to save the array at given offsets and in other GDAL compatible output formats.

Command Line Script

```
None provided.
```

Python Procedure

```
save(array_in, 'image_out' [,gis=] [,woffset=] [,hoffset=] [,format=])
```

Parameters

gis Optional GIS information to be saved to the image file header. This variable is returned from a call to the `load` function.

woffset *integer*, default None. If `woffset` or `hoffset` are set, it is assumed that `emptyfile()` has already been called to initialise the image size.

hoffset *integer*, default None. If `woffset` or `hoffset` are set, it is assumed that `emptyfile()` has already been called to initialise the image size.

format The following formats are supported:
VRT: Virtual Raster
GTiff: GeoTIFF
HFA: Erdas Imagine Images (.img)
NASoftware Ltd: *NASoftware Ltd NetCDF* Images (.nc)
SAR_CEOS: CEOS SAR Image
CEOS: CEOS Image
ELAS: ELAS
AIG: Arc/Info Binary Grid
AAIGrid: Arc/Info ASCII Grid
SDTS: SDTS Raster

DTED: DTED Elevation Raster
PNG: Portable Network Graphics
JPEG: JPEG JFIF
MEM: In Memory Raster
JDEM: Japanese DEM (.mem)
GIF: Graphics Interchange Format (.gif)
ESAT: Envisat Image Format (.N1)
BSB: Maptech BSB Nautical Charts
XPM: X11 PixMap Format (.xpm)
PNM: Portable Pixmap Format (netpbm)
DOQ1: USGS DOQ (Old Style)
DOQ2: USGS DOQ (New Style)
ENVI: ENVI .hdr Labelled
EHdr: ESRI .hdr Labelled
PAux: PCI .aux Labelled
MFF: Atlantis MFF Raster
MFF2: Atlantis MFF2 (HKV) Raster
EFF: Eosat Fast Format
FujiBAS: Fuji BAS Scanner Image
FIT: FIT image
USGSDEM: USGS Optional ASCII DEM
GXF: GeoSoft Grid Exchange Format

The default format is 'InfoSAR'.

B.3.18 `tojpeg`

Purpose

Converts image data to a JPEG bitmap, suitable for display in common graphics packages.

For each layer of the image, values that lies in the range `min` to `max` are mapped into the range 0 to 255. If `min` or `max` are not specified then the parameter `sd` is used with the following mapping:

$\mu - \texttt{sd} \times \sigma \rightarrow 0$
$\mu + \texttt{sd} \times \sigma \rightarrow 255$

where μ is the mean and σ is the variance of the data. A value of `sd` = 0 has the special meaning of

minimum $\rightarrow 0$
maximum $\rightarrow 255$

Complex data cannot be directly displayed as a bitmap and so must be converted to real data first. This is done by the option `cmpl2real`.

Command Line Script

```
tojpeg.py image_in image_out.jpg [cmpl2real=] [sd=] [min=] [max=]
```

`[verbose=] [tilesize=] [tmpdir=]`

Python Procedure

`See procedure makebitmap.`

Parameters

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:
"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

sd *float*, default 2.0.

min *list*, default None. This can either be a single value or a list. eg `min=0,3.7,-2` (Note no spaces allowed around the comma) or in python `min=(0,3.7,-2)`.

max *list*, default None. As `min` this can be a single value or a list of values.

verbose *integer*, default 0. If set to 1 then the estimated `min` and `max` values are printed out.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.3.19 `totiff`

Purpose

Converts image data to a TIFF bitmap, suitable for display in common graphics packages.

For each layer of the image, values that lies in the range `min` to `max` are mapped into the range 0 to 255. If `min` or `max` are not specified then the parameter `sd` is used with the following mapping:
$\mu - \texttt{sd} \times \sigma \rightarrow 0$
$\mu + \texttt{sd} \times \sigma \rightarrow 255$
where μ is the mean and σ is the variance of the data. A value of `sd` = 0 has the special meaning of
minimum $\rightarrow 0$
maximum $\rightarrow 255$

Complex data cannot be directly displayed as a bitmap and so must be converted to real data first. This is done by the option `cmpl2real`.

Command Line Script

```
totiff.py image_in image_out.tif [cmpl2real=] [sd=] [min=] [max=]
[verbose=] [tilesize=] [tmpdir=]
```

Python Procedure

```
See procedure makebitmap.
```

Parameters

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:

"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

sd *float*, default 2.0.

min *list*, default None. This can either be a single value or a list. eg `min=0,3.7,-2` (Note no spaces allowed around the comma) or in python `min=(0,3.7,-2)`.

max *list*, default None. As `min` this can be a single value or a list of values.

verbose *integer*, default 0. If set to 1 then the estimated `min` and `max` values are printed out.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.4 Image Information Routines

B.4.1 countseg

Purpose

Returns the number of segments in an image.

Command Line Script

```
countseg.py image_in
```

Python Procedure

```
segments = countseg(array_in)
```

B.4.2 gdalinfo

Purpose

A utility to give information about a GDAL (Geospatial Data Abstraction Library) supported dataset. Information provided includes geocoding informarion, image dimensions and data type.

Command Line Script

```
gdalinfo image_in
```

Python Procedure

```
None provided.
```

B.4.3 histo

Purpose

Writes the histogram of the input image to an ASCII file.

Command Line Script

```
histo.py image_in text_out [nbins=] [layer=] [min=] [max=] [sd=] [norm=]
[cmpl2real=]
```

Python Procedure

```
None provided.
```

Parameters

nbins *integer*, default 100. The number of data bins used to store the results.

layer *integer*, default *None*. If specified, select this layer from the input image.

sd *float*, default 2. The number of bins is spread over the interval $[\mu - \mathrm{sd} \times \sigma, \mu + \mathrm{sd} \times \sigma]$ where μ and σ are the mean and standard deviation of the data respectively. Any data values falling outside this range are placed into the bin with either the lowest or highest data values, as appropriate.

min *float*, default *None*. If specified, this sets the minimum histogram bin.

max *float*, default *None*. If specified, this sets the maximum histogram bin.

norm *float*, default *None*. If this parameter is specified, normalize the area under the histogram to this value.

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:
"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

B.4.4 `histogram`

Purpose

Provides a graphical display of the data distribution of a given image file. For multi-dimensional images, the histograms for individual layers can be viewed separately.

Command Line Script

```
histogram.py image_in [nbins=] [sdrange=] [norm=]
```

Python Procedure

```
None provided.
```

Parameters

nbins *integer*, default 100. The number of data bins used to sort the data.

sdrange *float*, default 2. The number of bins is spread over the interval $[\mu - \mathrm{sdrange} \times \sigma, \mu + \mathrm{sdrange} \times \sigma]$ where μ and σ are the mean and standard deviation of the data respectively. Any data values falling outside this range are placed into the bin with either the lowest or highest data values, as appropriate.

intensity *integer*, default 0. If set to 1 then the image is processed as an intensity image, and the data is not squared before calculating the statistics to work out the range to plot. If set to 0, then the range is calculated using statistics assuming its an amplitude image.

norm *float*, default None. If this parameter is specified, normalize the area under the histogram to this value.

B.4.5 `stats`

Purpose

Calculates a variety of statistics for a given image file. These include image dimensions, data type, minimum, maximum, mean, standard deviation and, for real data, looks measure and order parameter.

If the intensity flag is set to 0 (the default value), thus indicating the image contains amplitude data, then the statistics are calculated for the square of the data, i.e. the intensity image. The mean, minimum and maximum values are then squarerooted before being returned.

Complex data is converted internally to real data using the `cmpl2real` method before the statistics of the real data is displayed.

For multi-layer images the statistics from each layer is calculated.

The `stats` procedure returns a class containing the results. The script version prints the results to screen.

Command Line Script

```
stats.py image_in [whole=] [intensity] [width=] [height=] [cmpl2real=]
[calctex=] [tilesize=] [overlap=] [tmpdir=]
```

Python Procedure

```
s = stats(array_in [,whole=] [,intensity] [,width=] [,height=]
          [,cmpl2real=] [,calctex=])
print s
```

```
print 'Minimum =', s.min()
print 'Maximum =', s.max()
print 'Mean =', s.mean()
print 'SD =', s.sd()
print 'Looks Measure =', s.enl()
print 'Order =', s.order()
```

Parameters

whole *integer*, default 0. If `whole` is 1 then the looks measure and order parameter are calculated for the whole image rather than as the average of local estimates.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

width *integer*, default 5. Window width size for calculating local estimates of the looks measure and order parameters.

height *integer*, default 5. Window height size for calculating local estimates of the looks measure and order parameters.

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:

"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

calctex *integer*, default 1. Calculating the texture measures can be time consuming on large images. If you do not require this information, then running `stats` with `calctex=0` will save some time.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.4.6 `xgraphtool`

Purpose

Provides a graphical display of the data in a text file. The points for each graph should be listed on a single line. A new line represents a new graph.

Command Line Script

```
xgraphtool.py text_in
```

Python Procedure

```
None provided.
```

Parameters

axissize *integer*, default 10. Font size for axis labels.

legendsize *integer*, default 10. Font size for legend labels.

legend *integer*, default 1. Display legend.

x *list*, `x=min,max`. Min,max values for x axis.

y *list*, `y=min,max`. Min,max values for y axis.

title *string*, default *None*. Graph title.

xaxis *string*, default *None*. X axis label.

yaxis *string*, default *None*. Y axis label.

grid *integer*, default 1. Draw grid behind graph

colours *list*, default `colours=Red,Green,Blue`. Line colours (No spaces around commas allowed)

style *list*, default `style=solid,dot,long_dash,short_dash,dot_dash` Line drawing styles (No spaces around commas allowed).

thick *integer*, default 1. Line drawing thickness.

linespoints *list*, default `linespoints=square,dot,circle,cross,plus,triangle,triangle_down` Draw a marker at each data point. (No spaces around commas allowed)

markersize *integer*, default 1. Size of marker.

xrow *integer*, default *None*. Row of data file to use for x-axis points. If not specified use the values 0,1,2,...

yrow *integer*, default *None*. Rows of data file to use for y-axis points. Default is to use all rows of input file.

transpose *integer*, default 1. Transpose the data array so that points for the graph are read from each line of the file rather than along a single line.

save *string*, default *None*. Saves the plot to the specified filename. Use the extensions `.bmp`, `.xpm`, `.jpg`, `.png` to select file type.

Further Comments

Zooming controls with mouse:

- Left mouse double click - reset zoom.
- Right mouse click - zoom out centred on click location.
- Left mouse drag - Zoom box.

Command line options may be specified in a file (one per line) and used with the `--config` option. For instance `xgraphtool.py --config=pdf.ini` with `pdf.ini` containing

```
xrow=0
yrow=1
y=min
title=Histogram
yaxis=Count
xaxis=Deviations
grid=1
style=solid
thick=2
```

B.5 Classification Routines

B.5.1 `colclass`

Purpose

Converts a classification with regions labelled 0,1,2,... to a colour image for easier interpretation.

Command Line Script

```
colclass.py image_in image_out [format=] [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = colclass(array_in [,format=])
```

Parameters

format *string*, default *NASoftware Ltd* (*NetCDF*). The other useful format is `GTiff`.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.5.2 `classstats`

Purpose

Calculates the mean and order parameters for the masked regions of the original image. The mean and order parameters can then be used as supervised inputs for the `classify` routine.

The masked image is a single layer real valued image with the region labels corresponding to the training data for that class.

The class means should be a single layered real valued image of height equal to the number of layers. The number of classes is then assumed to be equal to the width of the given image.

The class orders should be a single layered real valued image of height equal to the number of layers. The number of classes is then equal to the width of the class orders image.

Command Line Script

```
classstats.py maskimage_in image_in meanimage_out orderimage_out
[cmpl2real=] [intensity=] [verbose=] [tilesize=] [overlap=] [tmpdir=]
```

Python Procedure

```
stats_class = class_stats(maskarray_in, imagearray_in, [,cmpl2real=]
                          [,intensity=] [,verbose=])
```

Parameters

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:

"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

verbose *integer*, default 0. If set to 1, prints out the class statistics.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

Regions labelled 1 in the mask file are assigned to class 0, regions labelled 2 are assigned to class 1 and so on. Regions labelled 0 are taken as the background and are not assigned to a class.
See also `mask_stats`.

B.5.3 `classmap2layers`

Purpose

Convert a classification image with regions labelled 0,1,2... to layers of 0/1 masks. Hence the pixels labelled 2 will be marked with a 1 in layer 2 of the output image.

Command Line Script

```
classmap2layers.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = classmap2layers(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

The individual layers of the output image can be obtained with the routine `split_layers`.

B.5.4 `layers2classes`

Purpose

Converts a set of 0/1 masks into a classification image with regions labelled 0,1,2,... The pixels in layer 0 will be given the label 0, the pixels in layer 1 the label 1 and so on.

Command Line Script

```
layers2classes.py image_in image_out [tilesize=] [tmpdir=]
```

Python Procedure

```
array_out = layers2classes(array_in)
```

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.5.5 `mask_stats`

Purpose

Calculates the mean and order parameters for the masked regions of the original image. The mean and order parameters can then be used as supervised inputs for the `classify` routine.

The masked image is a multi-layered real valued image with number of layers equal to the number of desired classes. Each layer has pixels coloured either 0 or 1, the latter denoting the spacial location of the training data for that class.

The original image either a single or multi-layered real valued image. The width and height of the original and masked images must be identical though they can differ in the number of layers.

The class means should be a single layered real valued image of height equal to the number of layers. The number of classes is then assumed to be equal to the width of the given image.

The class orders should be a single layered real valued image of height equal to the number of layers. The number of classes is then equal to the width of the class orders image.

Command Line Script

```
mask_stats.py maskimage_in image_in mean_out order_out [intensity=]
[cmpl2real=] [verbose=] [tilesize=] [overlap=] [tmpdir=]
```

Python Procedure

```
stats_class = mask_stats(maskarray_in, array_in [,intensity=] [,cmpl2real=]
                         [,verbose=])
```

Parameters

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:

"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

verbose *integer*, default 0. If set to 1, prints out the class statistics.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Further Comments

See also `classstats`.

B.5.6 mask_polstats

Purpose

Calculates the covariance matrix from a polarimetric image within the masked regions of 'image'. 'mask' is a 0/1 image where each layer indicates the training data for each class.

Command Line Script

```
mask_polstats.py maskimage_in image_in meanimage_out [tilesize=]
[overlap=] [tmpdir=]
```

Python Procedure

```
stats_class = mask_polstats(maskarray_in, array_in [,cmpl2real=]
                            [,verbose=])
```

Parameters

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:
"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs × abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

verbose *integer*, default 0. If set to 1, prints out the class statistics.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.5.7 make_classes

Purpose

Generates equally spaced class means from input data. If the `minimum` or `maximum` parameters are not given then the `minimum` and `maximum` parameters are estimated from the mean of the data plus or minus the `sd` spread.

Command Line Script

```
make_classes.py image_in image_out [num_classes=] [sd=] [minimum=]
[maximum=] [cmpl2real=] [intensity=] [verbose=] [tilesize=] [overlap=]
[tmpdir=]
```

Python Procedure

```
mean_array = make_classes(array_in [,num_classes=] [,sd=] [,minimum=]
                          [,maximum=] [,cmpl2real=] [,verbose=]
                          [,intensity=])
```

Parameters

num_classes *integer*, default 5.

sd *float*, default 2.0.

minimum *list*, default *None*.

maximum *list*, default *None*.

cmpl2real *string*, default `"abs"`. Chooses from a set of methods for converting the complex image to a real one:
"`abs`" absolute value.
"`mag`" magnitude of complex number (= abs $\times$ abs).
"`phase`" Phase of complex number.
"`real`" Real part.
"`imag`" Imaginary part.

intensity *integer*, default 0. Assume the data is amplitude (square-root intensity).

verbose *integer*, default 0. If set to 1, prints out the class means.

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

B.5.8 `matrix_combine`

Purpose

Combines classification means files (held in sets of images) into a single file.

Classification can be performed on either single or layered images. As single images can be com-

bined into a layered image with `layer_combine`, the class means files can be combined into a single file with `matrix_combine`.

Command Line Script

```
matrix_combine.py matrix1_in matrix2_in matrix3_in ...  matrix_out
```

Python Procedure

```
None provided.
```

Parameters

B.5.9 matrix_split

Purpose

Splits the classification means (held in means.nc) into a series of files. Results are written into the files ”output_000.nc”, ”output_001.nc”

Classification can be performed on either single or layered images. As layered images can be split into a set of single images with `layer_split`, the class means files can be split into a set of files with `matrix_split`.

Command Line Script

```
matrix_split.py means.nc output.nc
```

Python Procedure

```
None provided.
```

Parameters

B.5.10 matrix_print

Purpose

Displays the data in the means file, used by `classify`, in an easy to read form by class.

Command Line Script

`matrix_print.py means_in`

Python Procedure

`None provided.`

Parameters

tilesize *integer*, default 512. A tilesize of 512 x 512 is used.

overlap *integer*, default 48. To prevent obvious joins between tiles, an overlap of 48 is used.

tmpdir *directory*, default None. Intermediate results are written to a system temporary directory.

Appendix C

Library Interface

If your application supports external libraries, you can call *InfoPACK* routines directly. By using the library, you can use your own file formats, but since these routines operate on arrays in memory, none of the tiling code will be available.

C.1 Sample Code

In the `library` directory, there is a sample C program `segment.c` which shows how to call the three different segmenation routines in *InfoPACK*. and link in the library libinfolib.a. Under linux use

```
gcc -I. segment.c -L. -linfolib -lm
```

where `-lm` is the mathematics library. `-I.` says the header file `infopack.h` is in the current directory and `-L.` says the library `libinfolib.a` is also in the current directory.

C.2 Execution

To use the library you need to set the environment variable `InfoPACK` to point to the directory where the licence file (`infopack.lic`) is. For instance if the licence file was in the same directory as the executable, in DOS you could use

```
set InfoPACK=.
```

or in bash

```
export InfoPACK=.
```

This may have already been done when *InfoPACK* was installed.

If it works, when you run the executable you should get

```
Segment
........
MT Segment
.......
Polsegment
....
```

If you get the message

```
Segment
segment failed with error 5
```

then the environment variable `InfoPACK` has not been set properly.

C.3 C interface

All of the available C function calls are listed in the header file `infopack.h`. The routines accept "natural" arguments (`int`, `double`, `float*` pointers) so that there will not be any problems with automatic type conversion in C if the header file is not included.

For those routines that return an `int`, the return value will be one of:

```
INFOSAR_OK               /* 0 */
INFOSAR_NO_MEMORY        /* 1 */
INFOSAR_INVALID_DIMS     /* 2 */
INFOSAR_OUT_OF_RANGE     /* 3 */
INFOSAR_INVALID_ARGS     /* 4 */
INFOSAR_INVALID_LICENCE  /* 5 */
```

where

`INFOSAR_OK` The routine has successfully completed.

`INFOSAR_NO_MEMORY` There was an error `malloc`ing enough memory for the routine to complete.

`INFOSAR_INVALID_DIMS` The width or height is too small to process the data.

`INFOSAR_OUT_OF_RANGE` You have asked for an invalid combination of options. For instance running a median filter on segmented data.

`INFOSAR_INVALID_ARGS` One of the arguments passed to the routine has an invalid value.

`INFOSAR_INVALID_LICENCE` There was an error either in finding or reading the licence file.

C.3.1 Progress Function

Many of the *InfoPACK* functions accept a function argument

```
int (*progress) (int flag, const char *format, ...)
```

This `progress` routine is used to return error or information message back to the calling application. It is also occasionally called with `flag` set to `INFOSAR_PUR_BLIP` to indicate progress of the *InfoPACK* routine. Other values for `flag` are

```
INFOSAR_PUR_MEMERROR /* -3 */
INFOSAR_PUR_ERROR    /* -2 */
INFOSAR_PUR_WARNING  /* -1 */
INFOSAR_PUR_BLIP     /*  0 */
INFOSAR_PUR_MESSAGE  /*  1 */
```

where

`INFOSAR_PUR_MEMERROR` This is a warning message to report that the *InfoPACK* routine is unable to `malloc` enough memory.

`INFOSAR_PUR_ERROR` A fatal error has occured in the *InfoPACK* routine.

`INFOSAR_PUR_WARNING` A warning message.

`INFOSAR_PUR_BLIP` This is used to let the user know that the routine is progressing normally.

`INFOSAR_PUR_MESSAGE` A debugging or information message.

If you do not wish to use this feature, then `NULL` or `0` can be passed in instead of `progress`.

C.3.2 Complex Types

The C standard has only recently introduce a complex data type and so some C compilers do not fully supprort complex data. If you do not need complex data then you can

```
#define INFOPACK_NOCOMPLEX
```

before including `infopack.h` to omit those definitions that use complex data.

If you wish you call *InfoPACK* routines that accept complex data then you may need to include the `<complex.h>` header file and to change the line

```
/* You may change this to be your complex type */
typedef __complex__ float komplex;
```

If your compiler is ISO C99 complient, this could be changed to

```
#include <complex.h>
typedef _Complex float komplex;
```

Appendix D

NASoftware Conventions for *NetCDF*

The *NASoftware* file format, used as the default file format with *InfoPACK*, is based on the Unidata *NetCDF* (http://www.unidata.ucar.edu/packages/netcdf/) file format. *NetCDF* is a very flexible format that would allow something as simple as a single image to be described in many different ways. It would not be practicable for *InfoPACK* to be able to interpret all of these different descriptions and so we have defined a simplified format, based on *NetCDF*, known as the *NASoftware* format.

The *NASoftware* format allows for a single image to be stored in a file along with geographic (GIS) information. This single image may be comprised of several layers of images, but the type (e.g. short, float, complex) of each layer must be the same.

A *NetCDF* file is divided into four sections: dimensions, variables, attributes and data. The properties of the first three of these will now be described.

D.1 Dimensions

The *NASoftware Ltd* convention uses the following dimension names:

- `num_rows`
- `num_cols`
- `num_cols_complex`.

NetCDF does not provide direct support for complex data types. To get around this the names of the dimensions are used to indicate whether the data are real or complex. If both the columns dimensions are present they must satisfy:

$$\texttt{num\_cols\_complex} = 2 \times \texttt{num\_cols}.$$

D.2 Variables

A real-valued array is declared as

$$variable\_name(\texttt{num\_rows}, \texttt{num\_cols}).$$

The variable can have any of the standard *NetCDF* types (byte, short, int, float or double). Multiple layers can be stored in a single file, but all layers must have the same dimensions.

A complex-valued array is declared as

$$variable\_name(\texttt{num\_rows}, \texttt{num\_cols\_complex}).$$

Again, it can have any of the standard *NetCDF* types. The array contains `num_cols_complex` elements but there are only half that many (`num_cols`) complex elements; they are stored with the real and imaginary parts interleaved.

In summary: all integer types can be signed or unsigned; any type can be real or complex.

D.3 Attributes

Many of the attributes correspond directly to those used in the GDAL interface. These are indicated below. Where possible, the *NetCDF* tag used to represent the attribute is standard and all except the `Conventions` tag are optional.

D.3.1 Global Attributes

NetCDF Tag	`GDAL Member`	**Description**
`Conventions`		A character array containing the string `InfoSAR`.
`ConventionsURL`		A character array containing the string `href="http://www.nasoftware.co.uk"`
`geotransform`	`GeoTransform`	An array of six doubles corresponding to the GDAL geotransform parameters.
`projection`	`Projection`	A character array containing a projection description in one of the formats recognised by GDAL.

D.3.2 Variable Attributes

NetCDF Tag	GDAL Member	Description
long_name	Description	A character array containing a description of the data.
missing_value	NoDataValue	Any pixels in the data that have this value should be regarded as undefined and not part of the data. It is stored as a double.
classnames	CategoryNames	A character array containing a list of names that can be used as labels for the classes indexed by the image. The names are separated in the string by a newline (\n) character.
colourmap	ColorTable	An array of 4-byte integers - one per colour in the colour map. Each integer should be interpreted as four unsigned bytes; these represent red, green, blue, and opacity in the default (RGBA) interpretation. This can be changed by setting the palette tag. The colour map need only hold as many values as are indexed by the data. However, images translated from other formats may have a 256-entry table even though much of it is unused.
palette		A character array of length 4 containing one of RGBA, CMYK, HLSA or Gray. These describe how the colour map should be interpreted. The interpretation defaults to RGBA if this tag is not present.
display	ColorInterpretation	A character array of length 1 indicating how the layer should be shown on display hardware. The character stored is * for greyscale, K for black, or the first letter of one of the following terms: Red, Green, Blue, Alpha, Hue, Saturation, Lightness, Cyan, Magenta, Yellow. This tag should not be used when a colour map is present.